TOWARDS A MULTI-TIER RUNTIME SYSTEM FOR GIPSY

BIN HAN

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Concordia University

Montréal, Québec, Canada

MAY 2010

Your file Votre référence
ISBN: 978-0-494-71074-6
Our file Notre référence
ISBN: 978-0-494-71074-6

Abstract

Towards a Multi-Tier Runtime System for GIPSY

Bin Han

Intensional programming implies declarative programming, in the sense of Lucid, based on denotational semantics where the declarations are evaluated in an inherent multi-dimensional context space.

The General Intensional Programming System (GIPSY) is a hybrid multi-language programming platform and a demand-driven execution environment. GIPSY aims at the long-term investigation into the possibilities of Intensional Programming. The GIPSY's compiler, GIPC, is based on the notion of Generic Intensional Programming Language (GIPL) which solved the problem of language-independence of the runtime system by allowing a common representation for all compiled programs, the Generic Eduction Engine Resources (GEER).

In this thesis, we discuss the solution to GIPSY's Runtime System. The Multi-Tier framework which consists of Demand Generator Tier (DGT), Demand Store Tier (DST) and Demand Worker Tier (DWT), offers demand-driven, distributed execution and technology independent manners by integrating the previous research on the demand migration middleware implemented by Jini and Java Message Service (JMS).

Acknowledgments

I would like to thank my supervisor Dr. Joey Paquet for his ever lasting patience and caring guidance throughout the variety of the learning experience and the advices and insightful comments to make these contributions possible. I would also like to thank my friendly team members with whom we together are pushing the GIPSY project running forward. Specifically, I would like to mention Serguei A. Mokhov, Emil Vassev, Amir Pourteymour, Xin Tong, and Yi Ji for outstanding team work. Thanks to Dr. Jaroslav Opatrny for an in-depth introduction to compiler design.

Words are inadequate to thank my parents, Yuting Han and Hui Liu, and my sister Shu Zhang, for their unconditional love, support and immense source of inspiration for me all through my life.

This work has been sponsored by NSERC and the Faculty of Engineering and Computer Science of Concordia University, Montréal, Québec, Canada.

Contents

List of Figures

Chapter 1

Introduction

"Newton was a genius, but not because of the superior computational power of his brain. Newton's genius was, on the contrary, his ability to simplify, idealize, and streamline the world so that it became, in some measure, tractable to the brains of perfectly ordinary men."

Gerald M. Weinberg

1.1 Intensional Programming

One of the major challenges of computer science is the design of programming languages that would free the programmers from low-level, machine-related tasks. Such paradigms are usually based on mathematical foundations and they allow for a cleaner and more declarative way of programming. The *intensional* programming paradigm has its foundations in *intensional logic* [vB88] which is a branch of mathematical logic that has been used to describe context-dependent entities. The initial motivation for the development of intensional logic was the formal description of the meaning of natural language. Intuitively speaking, it is to develop a formal system for effectively describing entities whose value depends their implicit context of utterance. By extension, intensional programming aims at writing programs in which identifiers' evaluation is context-dependent, as well as allowing contexts to be first

class values.

Following this idea, one of the main characteristics of intensional programming is that it deals with infinite entities, as identifiers representing intensional entities can be evaluated in a potentially infinite number of contexts. These entities are treated as first class object by intensional languages. For example, two infinite streams of numbers can be added together, functions can be applied on infinite tables and trees, etc. Because of the above characteristic, intensional languages are especially appropriate for describing the behavior of systems that change with time or physical phenomena that use or depend on multidimensional contexts over potentially infinite dimensions such as time, three-dimensional space, temperature, etc. Such infinitely multidimensional entities can only be manipulated with difficulty with mainstream imperative programming languages that mostly rely on extensional iteration to compute data sets over such entities.

The infinite nature of intensional programming's identifiers suggests that their implementation might be problematic. For example, the output of an intensional program can be an infinite sequence. How can such a sequence be computed and delivered to the user? This is obviously not possible, as it would require infinite computation time. However, we can always expect to be able to compute larger and larger parts of the desired output of the program by *lazy evaluation* or *demand driven computation*, the technique of delaying a computation until a certain value of the output sequence is required. Thus, infinite entities are expressed in intensional programs, but their evaluation is made in a lazy, demand-driven fashion.

Lucid was the first intensional language developed. The most comprehensive description, its semantics, its applications, and its potential extensions is described in [WA85]. Since its inception, Lucid had been extended in several ways (an introduction about the Lucid family of languages is addressed in Section 2.2). Its variants have been used to specify three-dimensional spreadsheets [DW90a, DW90b], parallel computation models such as systolic arrays [Du91], attribute grammars [Tao94], real-time systems [FL89, PKL93], database systems [PP94] and *version control* [PW93]. The intensional versioning approach described

in [PW93] has found applications in the Internet. One example application in this domain is development of the language IHTML (Intensional HTML) [WBSY98], a high level web authoring language. The advantage of IHTML over conventional HTML is that it allows practical specification of web pages that can exist in many different versions. Each page of IHTML defines an intension - an indexed family of actual HTML pages which varies over a multi-dimensional author-specified version space. Authors can create multiple labeled version of the IHTML source for a given page. Requests from clients specify both a page and a version, and the server software selects the appropriate source page and uses it to generate the requested actual HTML page. Thus, authors do not have to provide separate source for each version. Web sites created by IHTML are easier to maintain and require significantly less space when compared to the sites created by cloning conventional HTML files.

The traditional implementation of Lucid programs is based on a computational model known as *eduction* [FW87]. The main characteristic of eductive computation is that it computes the value of expressions with respect to contexts. Thus, expression in a Lucid program may evaluate to a different value when evaluated in different contexts. This suggests that an efficient implementation of an intensional language should store values of variables that have been computed under specific contexts, so that these results will be available if demanded later during evaluation.

Among different implementations of programming platforms for Lucid variants, GLU introduced the concept of using the *eductive* evaluation model for distributed evaluation with *Remote Procedure Call* [Sun06]. Unfortunately, due to constant changes in Lucid languages, GLU was soon deprecated.

1.2 General Intensional Programming System

The General Intensional Programming System (GIPSY) [PW05, Lu04, Mok05, Mok10b, MP10] aims at providing a platform for the investigation on intensional programming and hybrid intensional-imperative programming. The GIPSY's compiler, General Intensional

Programming Compiler (GIPC), is based on the notion of Generic Intensional Programming Language (GIPL) [Paq99, PMT08, PMDW08, RP08], which is the core language into which all other flavors of Lucid can be translated. The notion of a generic language also solves the problem of language-independence of the run-time system by allowing a common representation for all compiled programs, the Generic Eduction Engine Resources (GEER). The GIPSY project is directed by Dr. Joey Paquet in the Computer Science & Software Engineering Department at Concordia University in Montreal, Quebec.

1.3 Problem Statement

GIPSY is a multi-language programming environment. It can compile and execute programs written in any variant of Lucid, and it allows Lucid dialects to use procedures written in imperative languages such as C, C++ or Java. The GIPSY compiler has been designed in a modular and flexible fashion that enables it to keep pace with the evolution of Lucid. At the same time, the distributed execution of hybrid program is another research direction. The original question that we planned to answer in this research work is the scalability of hybrid Lucid programs executed in a distributed demand-driven runtime system. Before we started, the implementation of the GIPSY run-time system was implemented using a GLU-like generator-worker distributed evaluation architecture, where:

- generators are implementing eduction engine and generating demands according to a Lucid program's declarations.
- workers are executing imperative procedures called by a hybrid Lucid program.
- a middleware layer is migrating demands and their resulting values over the distributed components, as well as storing them for further retrieval.

From the point of view of distributed evaluation, one of the most important such components is the middleware layer, which has been implemented as a *Demand Migration Framework* (DMF) [Vas05] which aims at providing generic middleware for demand migration for

distributed demand-driven evaluation of Lucid programs. Two different instances of this framework have been implemented using Jini and JMS [Vas05, Pou08]. However, these two implementations were using slightly different concepts for the notion of demand, and some of their key components do not share or inherit the same interfaces.

Another problematic aspect of our research goal is that, although our eduction engine is able to execute compiled programs locally, re-design is still needed for the distributed and demand-driven execution. It is important to note that the compiler design and program evaluation aspects fall outside of this thesis' research scope. This part of the work was done in collaboration with Serguei Mokhov, who is the main implementer of the GIPSY eduction engine. As an alternate solution to the implementation of the demand generator, we propose the implementation of a demand generation simulator (explained in Section 4.4.1), that enables a user to create demands out of a pre-fixed pool of different kinds of demands. This enables us to more accurately control the kinds of demands being propagated in the system, and will provide an appropriate environment to test for specific test cases such as for scalability testing. In the integration testing (more details in Section 4.4), we process all the three types of demands that are being propagated in GIPSY and inspect the demand status and their processing workflow.

Our general goal in this research thesis is thus to analyze, refactor, and re-design our existing GIPSY components so that they comply with our new ideas of demand-driven and distributed evaluation scheme.

1.4 Proposed Solution

The design architecture adopted (as shown in Figure 1) is a *multi-tier architecture*, where each of the three above-mentioned distributed evaluation components (generator, worker, middleware) are, respectively: the *Demand Generator Tier* (DGT), *Demand Worker Tier* (DWT) and *Demand Store Tier* (DST). In general, the DGT generates intensional demands and procedural demands according to the program declarations; the DST acts as a middleware between tiers in order to migrate and store demands, and the DWT is a tier that can

process procedural demands, i.e. a worker. Each one of these is wrapped into different tiers to perform the demand migration and execution task, and a workflow is defined to specify how they are collaborating to provide distributed demand-driven evaluation of hybrid Lucid programs. With this architecture, demands are propagated without knowing where they will be processed or stored, and any of the tier can fail without the system to be fatally affected. As stated in the problem statement, such components have been implemented by past members of the team, but do not necessarily mesh correctly together, and do not follow the same concepts for demand-driven evaluation. One of the objective of this thesis is constructing three sub-system which acts as an adapter by defining generic execution components to integrate the previously developed components into the GIPSY project to establish the runtime system which is a distributed demand-driven execution environment.

GIPSY is an evolving project, and the runtime system is related to different parts of the system. In order to carry out the proposed solution, we decided to apply *iterative and incremental development process*, key steps in the process are start with a simple implementation and then unit testing, at each iteration, design modifications are made and new functional capabilities are added.

1.5 Contributions

This thesis integrates the previous investigation of demand migration system with the generic eduction engine to implement GIPSY's runtime system. The main contribution of the thesis address the previously mentioned research problems through the following tasks:

- Analyze the workflow of the required runtime system following a unified demand-driven evaluation scheme.
- Study and refactor the previous implementation of DMSs, unify their interfaces to improve the design of existing code towards making the DMS implementations to be really interchangeable and in fact instances of our Demand Migration Framework (DMF).

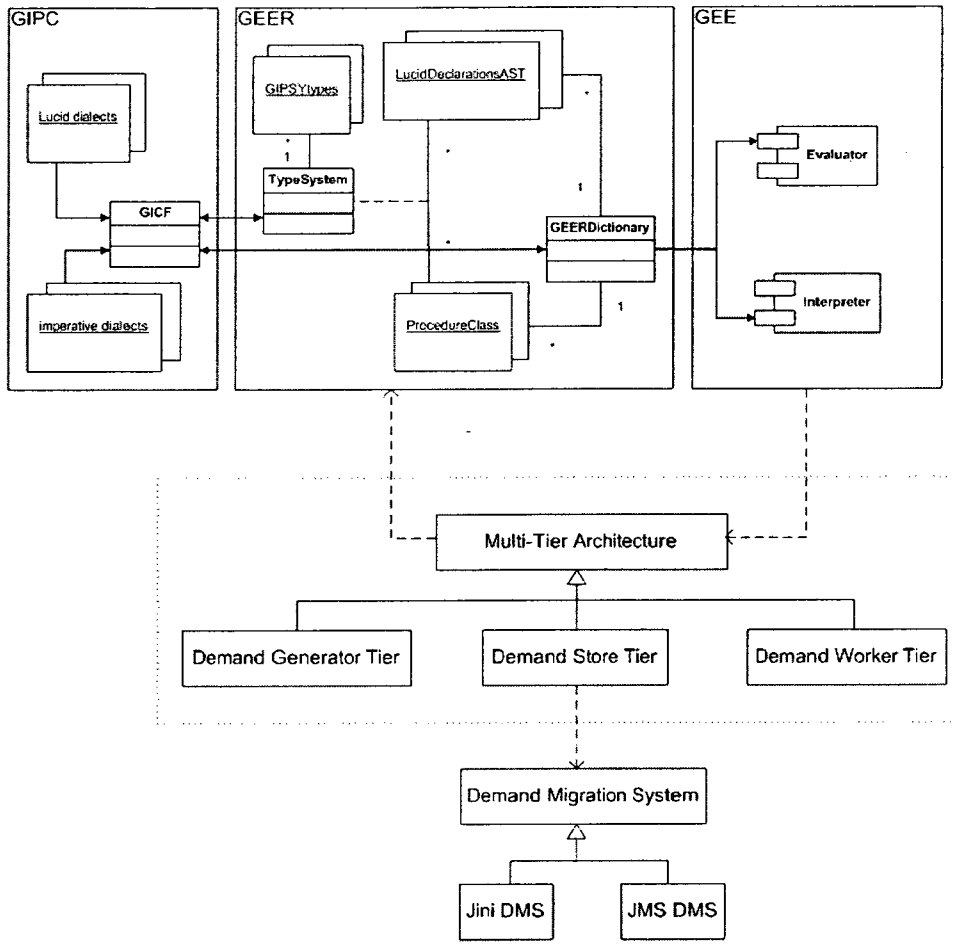

Figure 1: Multi-Tier Architecture within GIPSY

- Implement the multi-tier architecture that follows our latest ideas on demand-driven evaluation, in a manner that also offers loose coupling among different tiers which is very important for system integration and an efficient implementation.
- Unify the notion of *demand* as the internal migration object of GIPSY runtime system.
- Integrate the DMF into the multi-tier architecture, so that different DMF instances can be seemlessly interchanged.

1.6 Structure of the Dissertation

This thesis is organized in three parts. Part One (Chapter 1) introduces the reader to the problem domain and the proposed solution. Part Two (Chapter 2, 3 and 4) walks through the major aspects of the system design and implementation, and also the methodology that has been applied. Part Three (Chapter 5) covers the conclusion and future work.

- **Chapter 1:** This chapter answers the question of *"What"*, introduces the context of the problem and the proposed solution.
- **Chapter 2:** This chapter introduces the background of the research.
- **Chapter 3:** This chapter answers the question of *"Why"*, the rational of the design and the applied methodology are covered.
- **Chapter 4:** This chapter answers the question of *"How"*, describes the refactoring delivered, the implementation detail, and the corresponding testing.
- **Chapter 5:** This chapter makes the conclusion of the thesis and future work.

Chapter 2

Background

"The question of whether machines can think [...] is about as relevant as the question of whether submarines can swim. "

Edsger W. Dijkstra

The General Intensional Programming System (GIPSY) project is an ongoing effort aiming at providing a flexible platform for the investigation on the intensional programming model as realized by the latest versions of the Lucid programming language, a multidimensional context-aware language whose semantics is based on possible worlds semantics. GIPSY provides an integrated framework for compiling programs written in all variants of Lucid, even any languages of intensional nature that can be translated into Generic Lucid and also a demand-driven distributed execution environment. This thesis discusses our novel architecture framework for the run-time system that enables the distributed execution of such programs.

This chapter introduces the background of Intensional Logic, Intensional Programming, Lucid programming language, and the progress of the GIPSY project.

2.1 Intensional Logic

Intensional logic [Car47, DWP81, Gal75] comes from research in natural language understanding. Many sentences of the languages we use in everyday life are often ambiguous (i.e., they can be interpreted in different ways under different situation or from different people). This has led many scientists to believe that natural languages are not formal from a mathematical point of view, and it is, therefore, impossible to analyze and study them systematically. However, Montague (the father of intensional logic) firmly believe that natural language have a mathematical basis [DWP81] analogous to that of computer language. Intuitively speaking, the real meaning of a natural language expression whose truth-value depends on the context in which it is uttered is its *intension*. For example, we have an expression:

E: the average temperature is greater than $0°C$.

We can evaluate this expression in different contexts, for example: [`place:Montreal`, `month:January`], it adds *dimension* `place` and `month` to the expression and `Montreal` and `January` are the place holder *tags* along those dimensions. Dimension names mapped with respective tag values form the *context* in which this expression maybe evaluated.

We can compute a part of the extension of that intension in both the `place` and `month` dimensions. And the evaluation result of that expression will vary accordingly, as shown in the following: Given the `place` dimension tags $\{Montreal, Ottawa\}$, we have a valuation for E:

E =

	Ja	Fe	Mr	Ap	Ma	Jn	Jl	Au	Se	Oc	No	De
Montreal	F	F	F	F	T	T	T	T	T	F	F	F
Ottawa	F	F	F	F	F	T	T	T	F	F	F	F

Generalizing the above discussion, we can say that in many cases, the meaning of a natural language expression is a function from contexts to values. Such a function is called the *intension* of the expression. The value of the intension at a particular context is called the *extension* of the expression at that context.

Except this context-dependent character, another characteristic of natural language is that they use *context-manipulation operators* to alter the present context. For example, we say "tomorrow's temperature" in order to refer to the value of the temperature the next day. Words such as *west*, *next* are often used to change the context. Using these operators, we avoid referring to the new context explicitly, but we specify it as a function of the present context.

However, if intensional logic is close to real language, why not use it in order to design new, more powerful and expressive programming languages? This topic is discussed in the next section.

2.2 Lucid Programming Language

Most existing programming languages are machine oriented: the programmer is ofter required to know many aspects of the architecture of the underlying machine in order to be able to write even the simplest programs. Consider, for example, the notions of *variable* and *assignments* in an imperative language. Programmers usually think of *variables* as memory locations and *assignments* as commands that alter the content of such locations. Therefore, we are interested in a programming paradigm that would hide unnecessary operational concepts from the programmer, in the same way that a natural language helps people hide many unnecessary context details during everyday talk. One approach is to develop programming languages based on intensional logic. Now we present a simple intensional program Listing 2.1 that computes the infinite sequence of all nature numbers:

```
result = nature;
nature = 0 fby (nature + 1);
```

Listing 2.1: Infinite Sequence of Nature Numbers

Note the use of the operator FBY (read *followed by*) in the above program. For the time being, we provide the intuitive reading of the program:

The result of the program is the sequence of natural numbers. The first value of the natural

number sequence is 0. The next value in the sequence can be produced by adding 1 to the current value of the sequence.

Note that the usual mathematical definition of the sequence of natural numbers involves the use of a *time index*:

$nat_0 = 0$

$nat_{t+1} = nat_t + 1$

The intensional program avoids the explicit use of the time index. The sequence of the natural numbers is defined using the temporal FBY rather than using subscripts.

The above example is actually a program written in the intensional language Lucid. Lucid was originally designed as a *dataflow programming language.* Upon defining its semantics formally and generalizing it to the multidimensional case, Lucid went to be called an *indexical programming language.* In fact, intensional programming may also be called *multi-dimensional programming* because the expressions involved are allowed to vary in arbitrary number of dimensions, the context of evaluation being a multi-dimensional context. In order to present the unusual evolution of Lucid, we present the following list of programming languages with the evolution of Lucid dialects:

- **Lucid**, 1974-1977 [AW76, AW77a, AFJW95]: Lucid is initially designed to experiment with non-von Neumann programming models by Edward A. Ashcroft and William W. Wadge through 1974-1977. At the beginning, it was a dataflow programming language for natural expression of iterative algorithms [AW77b]. For example, FBY is one of the basic operators from the original Lucid, it stands for "followed by", defines what comes after the previous expression.

  $X = 1 \text{ FBY } X + 1$

  In Lucid, each variable defines an infinite and multi-dimensional stream of values, the last equation defines a stream X using the FBY operator. In this instance, the equation defines a sequence *(1, 2, 3, ..., i, ...)*.

- **Indexical Lucid**, 1996 [JD96, JDA97]: Prior to Indexical Lucid, the only implied dimension was the time dimension, tagged with a set of natural numbers. With Indexical Lucid, we can have more than one dimension, and we can query for a part of the context. Thus,

the syntactic definition has been amended to include an ability to specify which dimensions exactly we are working on. And also, it is in Indexical Lucid that the # and @ operators were defined. It was found that all Lucid operators could be expressed in terms of # and @.

- **Granular Lucid** (GLU), 1996 [JD96, JDA97]: First hybrid intensional-imperative paradigm (C/Fortran and Indexical Lucid), where a GLU program is defined in two parts: an Indexical Lucid part acting as a *skeleton language*, into which user-defined functions written in the second part (written in C or Fortran) are called. This model was invented as a proof-of-concept to demonstrate the dataflow model of computation attributed to Lucid at the time. Allowing Lucid to call procedures written in a standard procedural language allowed for increased granularity of computation, identified as a flaw when executing fine-grained dataflow programs defined by standard "pure Lucid" programs. In the cases where the nature of the procedures allowed them to be executed in parallel, this model also allowed GLU to be used as a "program parallelization" platform [JDA97]. The tagged-token demand-driven dataflow model (called *eduction*) based on intensional logics used in the evaluation of Lucid programs did also provide a failure-resistant model of distributed/parallel computation. The GLU model is the basis of the design of the GIPSY.

- **Generic Intensional Programming Language** (GIPL), 1999 [Paq99]: All Lucid dialects can be translated into the basic form of Lucid, GIPL, through a set of translation rules. GIPL is in the foundation of the execution semantics of GIPSY and its compiler (GIPC) and execution engine (GEE) because its AST is the only type of AST the engine understands when executing a GIPSY program. Figure 2 shows the syntax of GIPL.

- **Lucid Enriched With Context** (Lucx), 2003-2005 [WAP05, Wan06]: Kaiyu Wan introduced the notion of context as first class value in Lucid, thus context can be declared and manipulated directly in Lucid programming language. She also provides a set of well-defined context calculus operators performed on context values to yield new contexts for different applications. Context calculus has been implemented by Xin Tong in [Ton08].

- **JLucid Objective Lucid**, 2003-2005 [Mok05, GMP05]: JLucid brings embedded Java and most of its powers into Indexical Lucid into the GIPSY by allowing intensional languages to manipulate Java methods as first class values.

$$
\begin{array}{rcl}
E & ::= & id \\
 & | & E(E_1, ..., E_n) \\
 & | & \textbf{if } E \textbf{ then } E' \textbf{ else } E'' \\
 & | & \#E \\
 & | & E@E'E'' \\
 & | & E \texttt{ where } Q \\
Q & ::= & \texttt{dimension } id \\
 & | & id = E \\
 & | & id(id_1, id_2, ..., id_n) = E \\
 & | & QQ
\end{array}
$$

Figure 2: GIPL Syntax

- **Object Oriented Intensional Programming Language** (OOIP), 1994-present [WPM07, WPM08, WPM10]: Object Oriented Intensional Programming Language combines the essential characteristics of Lucid and Java. It introduces the notion of object streams which makes it possible that each element in an intensional stream can be an object with embedded intensional properties. At the same time, it also brings Java objects the power to express context, creating the novel concept of *intensional objects* on context-aware objects.

2.3 Eductive Model of Computation

Eduction can be described as "tagged-token demand-driven dataflow" computing. The central concept to this model of execution is the notion of *generation*, *propagation*, and *consumption* of *demands* and their resulting values. Lucid programs are declarative programs where every identifier is defined as an expression using other identifiers and an underlying algebra. An initial demand for the value of a certain identifier is generated, then the eduction engine, using the defining expression of the identifier, generates demands for the constituting identifiers of this expression, on which operators are applied in their embedding expressions. These demands in turn generate other demands, until some demands eventually evaluate to some values, which are then propagated back in the chain of demands, operators are applied to compute expression values, until eventually the value of the initial demand is computed and returned.

The GIPSY uses eduction model of computation based on the principle that certain computation takes effect only if there is an explicit demand for it and every computed value is placed

in the *warehouse* [Tao04], and every demand for an already computed value is extracted from the warehouse rather than computed another time. Eduction can be understood as a two-way traffic in the communication lines. In one direction value flows from producers to consumers, in the usual way. In the other direction, demands are issued from the consumers to the producers if the original demands needs any further computation, which reduces the overhead induced by the need for procedural demands computation.

2.4 General Intensional Programming System

The *General Intensional Programming System* (GIPSY) [PK00, PW05] is a multi-language programming platform and demand-driven distributed execution environment. It aims at the long-term investigation into the possibilities of the intensional programming model as realized by the latest versions of Lucid. It provides an integrated framework for the programs written in all variants of Lucid, and even any other language of intensional nature that can be translated into *Generic Lucid* (GIPL). The GIPSY project is directed by Dr. Joey Paquet in the Computer Science & Software Engineering Department at Concordia University in Montreal, Quebec.

GIPSY Framework The GIPSY framework consists of three modular sub-systems: *the General Intensional Programming Compiler* (GIPC) [Ren02, PGW04], *the General Eduction Engine* (GEE) [Mok05, MP05] and *the Multi-Tier Runtime System* (MTRS) [Vas05, VP05b, Pou08].

In the GIPSY, a *GIPSY Program* may consist of two parts: the *Lucid* part that defines the intensional data dependencies between identifiers and, optionally, the *sequential* part that defines the granular sequential computation units (written in a procedural language). The GIPSY program is compiled in a two-stage process, as depicted in Figure 4. First, the intensional (GIPL) part of the GIPSY program is parsed, and then translated in Java data structures, then the resulting Java program is compiled in the standard way, resulting in runtime system resources that we call a GEER (General Eduction Engine Resources). The GEE (General Eduction Engine) functions as either an interpreter that uses the GEER to execute the program locally or as an evaluator that generate demands based on he GEER for later distributed execution by the Multi-Tier Runtime System.

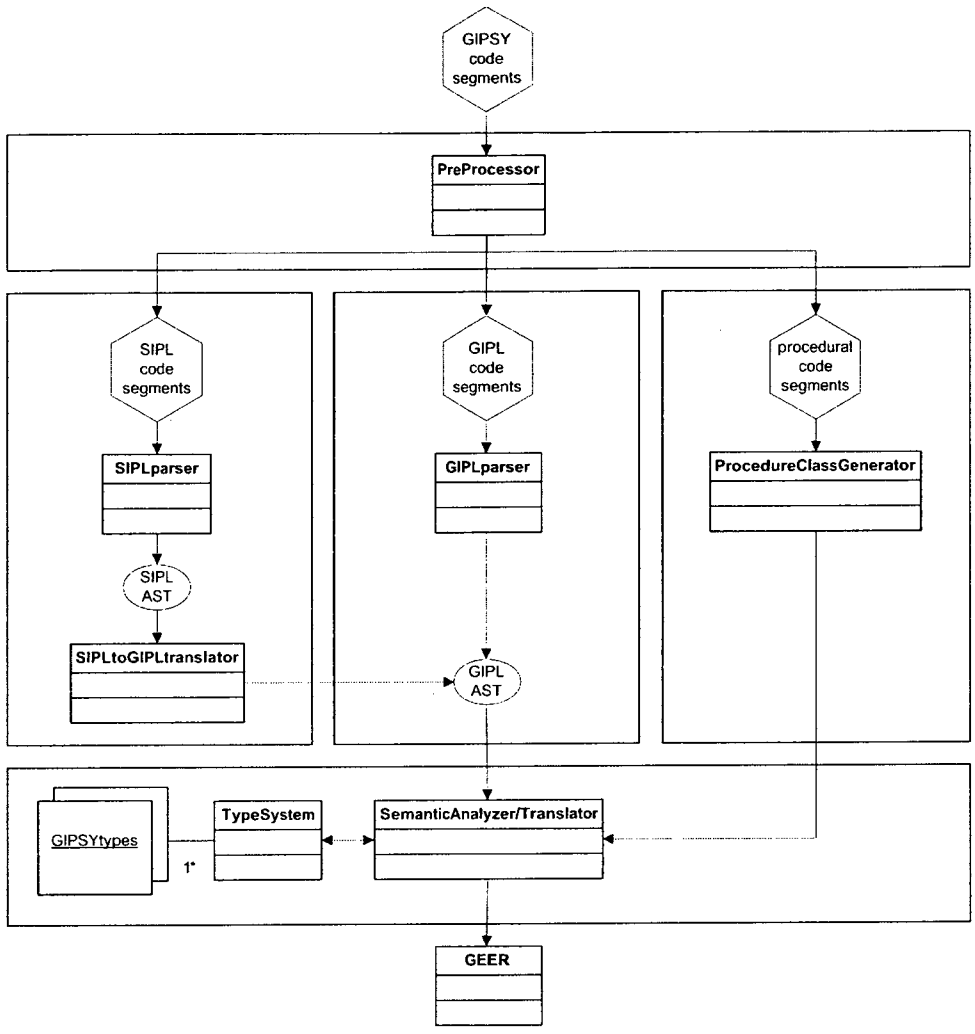

Figure 3: General Intensional Programming Compiler Framework

General Intensional Programming Compiler The GIPC in Figure 3 provides a generic infrastructure that enables the compilation of hybrid programs in Listing 2.2 written using various dialects of Lucid, as well as various procedural programming languages, in which the Lucid(intensional) parts are calling procedures written in these various procedural languages. One of the main precepts of this framework design is that of the GIPL being a generic language into which other Lucid dialects can be translated. This precept has two main goals: (1) to make the intensional part of the compiler framework easy to extend and (2) the runtime system is independent from the compiler, which means the GIPSY back end will not be affected each time a new Lucid dialect is introduced.

```
/**
 * Language-mix GIPSY program.
 * @author Serguei Mokhov
 */
#typedecl

myclass;

#funcdecl

myclass foo(int,double);
float bar(int,int):"ftp://newton.cs.concordia.ca/cool.class":baz;
int f1();

#JAVA
myclass foo(int a, double b)
{
   return new myclass(new Integer((int)(b + a)));
}

class myclass
{
   public myclass(Integer a)
   {
      System.out.println(a);
   }
}

#CPP
#include <iostream>

int f1(void)
{
   cout << "hello";
   return 0;
}

#OBJECTIVELUCID

A + bar(B, C)
where
   A = foo(B, C).intValue();
   B = f1();
   C = 2.0;
end;

/*
 * in theory we could write more than one intensional chunk,
 * then those chunks would evaluate as separate possibly
 * totally independent expressions in parallel that happened
 * to use the same set of imperative functions.
 */

// EOF
```

Listing 2.2: Example of a hybrid GIPSY program.

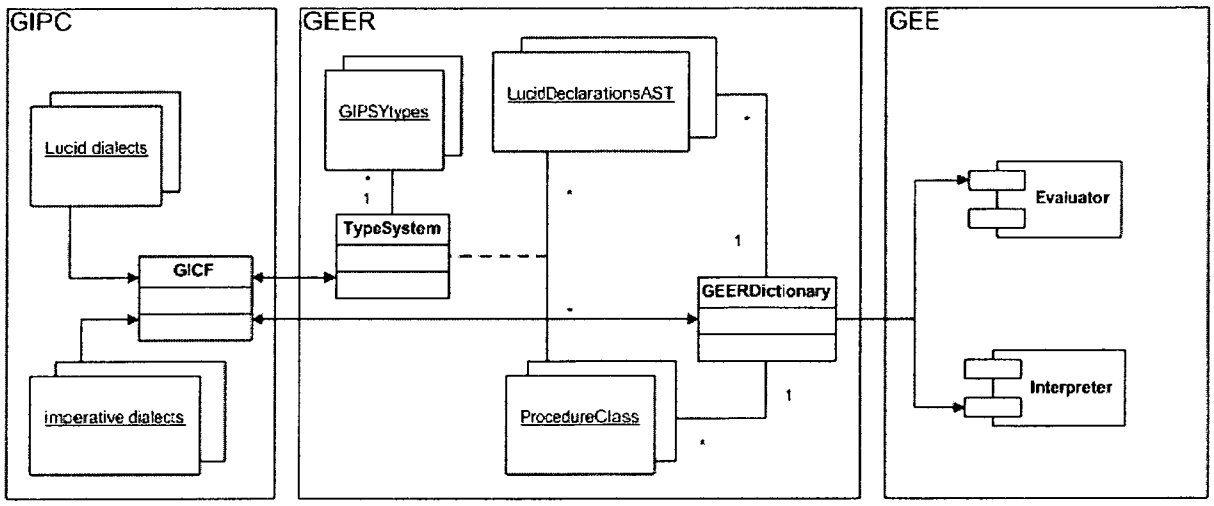

Figure 4: Compilation/Execution of GIPSY Programs

Generic Eduction Engine Resource *Language independence* of the runtime system is one of the central concepts. To achieve that, we rely on an intermediate representation which is generated by the compiler: the Generic Eduction Engine Resource (GEER) in Figure 4. The GIPC compiles a program into an instance of the GEER, a dictionary of identifiers [PGW04]. The framework of GIPC provides the potential to allow the easy addition of any flavor of the Lucid language through automated compiler generation taking semantic translation rules in input [Wu02].

Generic Eduction Engine The GEE is composed of three main modules: the executor, the intensional demand propagator, and the intensional warehouse. First, the GEER is fed to the demand generator by the compiler (GIPC). The demand generator receives the initial demand, that in turn raises the need for other demands to be generated and computed as the execution progresses. For all non-functional demands, the demand generator makes a request to the warehouse to see if this demand has already been computed. If so, the previously computed value is extracted from the warehouse. If not, the demand is propagated further, until the original demand resolves to a value and is put in the warehouse for further use. This type of *eduction* computation model was introduced by GLU 2.2 due to its distributed nature but can certainly be applicable to any functional language to improve efficiency.

Bo Lu was the first one to do the original design of the GEE framework [Lu04] and investigate its performance under threaded and distributed environments. Next, Lei Tao contributed the first

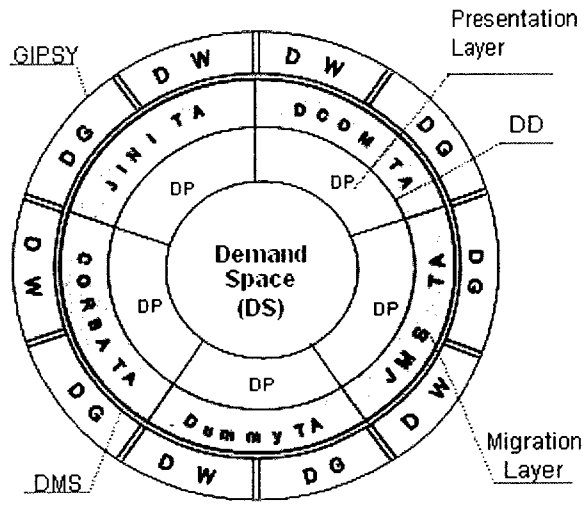

Figure 5: Demand Migration System Layered Structure

incarnation of the intensional value warehouse and garbage collection mechanisms in [Tao04]. Further, Emil Vassev [Vas05] and Amir Hossein Pouteymour [Pou08] had produced two general and functional *demand migration systems.*

Demand Migration System *Demand Migration System* (DMS) is a generic schema for migrating demands in a heterogeneous and distributed environment, it establishes a context for performing demand migration activities, where the migrated entities encapsulate embedded functions and data pertaining to the processing of these demands. Now we have two DMS [PVP07] based on the distributed technologies of JINI [Vas05] and JMS [Pou08]. A layered structure (Figure 5 [VP05b]) is applied in the design and implementation of DMS.

The DMS consists of two principal functional layers called *Demand Dispatcher* and *Migration Layer.* The Demand Dispatcher is an object storage mechanism able to dispatch the objects to their recipients. The Migration Layer is the layer preforming the object migration from the Demand Dispatcher to the recipient GIPSY workers or generators. These two DMSs middleware provides the demand transportation mechanism for the runtime system of GIPSY.

2.5 Related Work

GLU The first operational model for computing Lucid programs was designed independently by Cargill at the University of Waterloo and May at the University of Warwick based on Kripke models and possible-worlds semantics [Kri66, Kri69]. This technique was later extended by Ostrum for the implementation of Luthid interpreter [Ost81]. Luthid being tangential to standard Lucid, its implementation model was later used as a basis to the design of the pLucid interpreter by Faustini and Wadge [FW87]. Based on the same model, GLU was the most general intensional programming tool recently available [JD96]. However, with the continual evolution of Lucid language, the lack of flexibility and adaptability becomes the first concern of GLU [Paq99], despite its being very efficient. GLU's need of successor delivered two projects: GLU# and our GIPSY project.

GLU# GLU# [PK04] is a successor of GLU, which enables Lucid within C++. The authors argue for the embedding of small functional/intensional-language pieces of Lucid into C++ programs allowing lazy evaluation of arrays and functions thereby making Lucid easily accessible within a popular imperative programming language, such as C++. However, it suffers from the same inflexibility as GLU did and targets only C++ as a host language.

2.6 Summary

In this chapter, we introduced the background knowledge of this research, including: *intensional logic*, *Lucid programming language*, *eductive model of computation*, *General Intensional Programming System*, and the *related work*. In the next chapter, we discuss the design work of the research.

Chapter 3

Design & Methodology

"Computer science is the discipline that believes all problems can be solved with one more layer of indirection."

Dennis DeBruler

In this chapter, we introduce some of the key concepts in GIPSY that affect our design decisions, in Section 3.2 we overview the *Multi-Tier Architecture* that we are implementing, and in Section 3.3, we discuss the existing *Demand Migration System* that will be wrapped into the multi-tier framework. At last, we present the methodology that has been applied to carry out the implementation.

3.1 Key Concepts in GIPSY

Since this thesis stands inside an existing research and development project, our design and implementation is constrained by the existing concepts, theory, and code base. In this section, we aim at explaining such design issues used in the GIPSY project, following the two main aspect that concern us in the implementation of a run-time system for GIPSY:

- The necessity for GIPSY to rely on a distributed runtime system is discussed in Section 3.1.1 and Section 3.1.2.
- Eductive computation is the execution model of multi-tier runtime system design. It is introduced in Section 3.1.4 to describe the internal workflow of the runtime system.

3.1.1 Lucid as Dataflow Programming Language

Lucid was designed by Bill Wadge and Ed Ashcroft and introduced through 1974-1977 [AW76, AW77a]. Originally, it was a pure dataflow programming language using a tagged token demand-driven computational model. In this model, each statement can be understood as an equation defining a network of computing units and communication lines between them through which data flows. Each variable is an infinite stream of values and each function is a filter.

The real difference between Lucid and other languages is the way of modularization. In Lucid, the basic concepts are those of "stream" and "filter". A module in Lucid is therefore a filter and the hierarchical approach involves building up complicated filters by combining simpler ones. To summarize, Lucid programs could always be drawn as a dataflow chart, while others such as imperative language usually be drawn as a control flow chart.

Here is a simple dataflow example for *the Hamming Problem* (introduced by Richard Wesley Hamming) which requires to generate in ascending order all numbers divisible only by a given set of primes, usually 2, 3 and 5. It is a standard problem with standard solutions, the solution types vary according to the language of the programmer. The problem is easy to state, but for the imperative programmer it is not at all obvious to proceed. Usually it requires three nested loops to generate an array filled with the "Hamming numbers" but unordered, sorting is needed afterwards. But for Lucid, there is a very easy dataflow solution, and the solution can be expressed naturally and elegantly [WA85, Wad03].

```
/* Hamming Problem
*/
   h
        where
         h = 1 fby merge(merge(2*h,3*h),5*h);
         merge(x,y) = if xx <= yy then xx else yy fi
         where
          xx = x upon xx <= yy;
          yy = y upon yy <= xx;
         end;
        end
```

Listing 3.1: Lucid Solution for Hammings Problem

The Lucid solution in Listing 3.1 is based on the following observation: if h is the desired stream, then the streams $2 * h$, $3 * h$ and $5 * h$ (the streams formed by multiplying the components of h by 2, 3, and 5 respectively) are substreams of h. Furthermore, the values of the stream h are

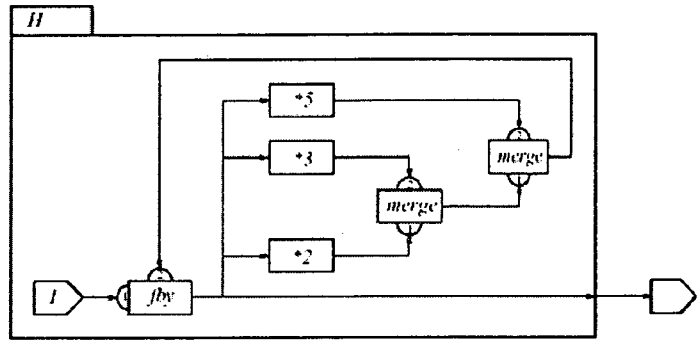

Figure 6: Lucid Dataflow Diagram

1 followed by the result of merging the streams $2 * h$, $3 * h$ and $5 * h$. The *merge* filter (function) produces the ordered merge of its two arguments. The dataflow graph looks like Figure 6 [Wan06].

From the above example we can see that a Lucid program can be easily regarded as filters function "coroutine" [AW82] which falls in the model of parallel computing paradigm.

3.1.2 Hybrid Programming

Lucid being a descendant of Landin's ISWIM [Lan66], it is defined as being independent of data types and algebra applied to the values it manipulates. Typically, the algebra embedded in Lucid is the standard arithmetic algebra. Inherently, due to their semantics, programs in this family of programming language (i.e. functional programming languages) can be evaluated in parallel. For example, given the definitions:

$A = B + C;$

$B = 2;$

$C = f(...) + D;$

$D = 4;$

since the definition of B and C do not have dependencies, they can be evaluated separately on different processing units. However, if B and C are integer values, for example:

$A = B + C;$

$B = 2;$

$C = 3;$

the benefits of distributed evaluation are clearly non-profitable due to the communication overhead. Thus, in order to profitably exploit the inherent parallelism of Lucid program execution, we have to find a way to augment the granularity of the algebra embedded in Lucid by incorporating aggregate data types that would allow Lucid to manipulate the corresponding operators that Lucid expressions would use.

The first attempt at this idea was made in the GLU project, by Jagannathan and Dodd at SRI in the 1990s [Agi95, JD96, JDA97]. GLU was in fact a hybrid Lucid/C language that allow a Lucid program to use an algebra defined in the C language. In a GLU program, a Lucid program is defined that manipulates data elements whose type is defined using the *C struct* syntax, and functions are defined in C syntax that represent the operators of the algebra used by the Lucid program.

3.1.3 Context in GIPSY

As mentioned in Section 2.1 and Section 2.2, a *context* is a point of reference in the multi-dimensional space in which an identifier is evaluated. A theory of contexts, including context calculus was designed by Wan in [Wan06], and implemented by Tong in [Ton08]. Referring to the definition of *demand* in intensional programming, a demand is a request for the evaluation of a certain identifier (i.e. a Lucid identifier, or a procedure identifier embedded in a hybrid Lucid program) in a given multi-dimensional context. Demands can be either *intensional* or *procedural* demands. The notion of demand is extended to include other types of demands related to the system's operation (systems demands), more detail about demands will be described in Section 3.2.3.

3.1.4 Eductive Computation

Eduction was first implemented in the pLucid interpreter. Later on, GLU supported a similar so-called "tagged-token demand-driven dataflow" computational model where data elements are computed on demand following a dataflow network defined in Lucid. In this model, data elements flow in the normal flow direction and demands flow in the reverse order, both being tagged with

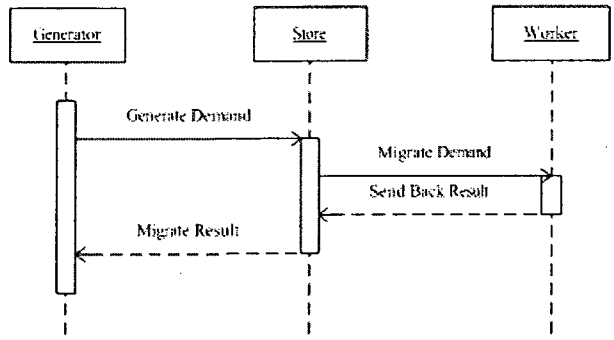

Figure 7: General Architecture

their context of evaluation. Eduction is a concept that will be easier to describe with an example. Consider the program in Listing 3.2 which outputs the stream *1, 4, 9, 16, ...* of squares of natural numbers. The first value should be produced for the whole program is the value of stream j at time 0, whose value is 1. The next value in need is that of the program at time 1, with the same idea, the value of j at time 1. The definition of j tells us that the result is 4.

$$j_{time=1} = 1 \text{ FBY } j_{time=0} + 2 * i_{time=0} + 1$$

as we know, $j_{time=0} = 1$ and $i_{time=0} = 1$, so the final result of $j_{time=1}$ is 4.

```
/* Sample for Eduction
*/
   j
    where
     i = 1 fby i+1;
     j = 1 fby j + 2*i + 1;
    end
```

Listing 3.2: Sample for Eduction

The basic principle of the eductive model should be apparent: the entire computation is driven by the attempts to compute the variable at different times. Eduction can be understood as a form of two-way traffic communication: in the usual way, the *generator* generates the demand for the value of variable at certain time, in the other way, the result is sent back by the *worker*. Furthermore, a storage mechanism for the input and output line of generator and worker is required. So far, the solution for GIPSY runtime system gets into shape, which is a three component architecture, generator and worker with a store module which will migrate the *demands* and their corresponding results. Figure 7 shows the general architecture.

3.1.5 Demand in GIPSY

As mentioned in Section 2.1 and Section 2.2, a *context* is a point of reference in the multi-dimensional space in which intensional expressions are evaluated. Referring to the definition of *demand* in intensional programming, a demand is a request for the evaluation of a certain identifier (i.e. a Lucid identifier, or a procedure identifier embedded in a hybrid Lucid program) in a given multi-dimensional context. It can be either *intensional* or *procedural* demand. The notion of demand is extended to include other types of demands related to the system's operation, more detail about demand will be described in Section 3.2.3.

From the above statements, we can get such conclusion: the denotational semantics [AW82] of Lucid and GIPSY's hybrid programming capacity require the runtime system to operate in a distributed manner; and the eductive model of computation enhances the generator-worker architecture developed in earlier solutions. In Section 3.2, we will introduce how we turn the requirements into the actual design.

3.2 Conceptual View of Multi-Tier Runtime System

3.2.1 Design Rationale

A statement of the philosophy of *modular programming* can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [GP70], which we quote below:

> *A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion, system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.*

The benefits expected of modular programming are: (1) Modules can be written with little knowledge of the code of other modules. (2) Modules can be reassembled and replaced without

reassembly of the whole system. (3) The system can be understood one module at a time. This is our rationale for dividing the runtime system into separate modules. The generator module will analyze the Lucid program and issue demands for the computation, the worker module will take care of functional demand computation.

From the research work done by J. Waldo, G. Wyant, A. Wollrath and S. Kendall, because the need of being aware of latency, memory access, partial failure and concurrency, the objects that interact in a distributed system intrinsically different from objects that interact in a single address space [WWWK94].

Given the above two theories, we decided to divide the whole system into different components, the distributed storage subsystem that takes the role of warehouse and demand migration will be isolated from modules which locally generate demands and compute value. Even further, we make the storage module as a middleware, such mechanism can simplify the development and also reduce the complexity of dealing with extensively distributed system. More information about the middleware approach will be discussed in Section 3.3.1.

3.2.2 Architecture Design

The design architecture adopted is a distributed multi-tier architecture, where each tier can have any number of instances, executing on a network of distributed computation nodes that can host any number of such tiers. The following sections introduce central concepts to the operation of this distrubuted execution architecture.

Generic Eduction Engine Resource

The *Generic Eduction Engine Resource* (GEER), first introduced in [Mok05], is an intermediate representation that is generated by the General Intensional Programming Compiler (GIPC) and will be processed by the Generic Eduction Engine (GEE) of GIPSY which will be expanded by the Multi-Tier Runtime system. GEER is our solution to achieve language independence of the runtime system. Figure 4 describes the role of GEER.

As the name suggests, the GEER structure is generic across all Lucid variants, in the sense that the structure and semantics of the GIPSY is independent of the Lucid variant in which its corresponding source code was written. This is necessitated by the fact that the engine was designed

to be "source language independent", one of the most important features offered by the presence of GIPL which is introduced in Section 2.2 as a generic language in the Lucid family of languages. Thus, the GIPC first translates the source program that might be written in any flavor of Lucid into "generic Lucid", then generate the GEER for this program, which is then made available at runtime to the various tiers executing this program.

The GEER contains all Lucid identifiers in a given program, typing information, rank i.e. dimensionality information, as well as an abstract syntax tree representation of the declarative definition of the identifiers in this program. It is the latter tree that is later on traversed by the demand generator in order to proceed with demand generation. In the case of hybrid Lucid programs, the GEER also contains a dictionary of procedures called by the Lucid program, known as *Procedure Classes*, as they in fact are wrapper classes wrapping procedures inside a Java class in cases where the functions being called are not written in Java [Mok05].

GIPSY Tier

The solution proposed for the runtime system of GIPSY is a Multi-Tier Architecture where the execution of GIPSY programs is divided into different tasks that are assigned to separate *tiers*. Each GIPSY tier is a separate process that communicates with other tiers with demands, which makes the evaluation model being called *demand-driven*. The demands are generated by the tiers and migrated to other tiers by the store tier. In this thesis, we refer to "tier" as an abstract and generic entity that represents a computational unit that is independent from other processes and collaborates with each other to achieve program execution. We will have a brief introduction about each tier in the following sections, the internal working mechanism of each tier will be explained in Chapter 4.

GIPSY Node

A GIPSY *node* is a computing unit that hosts one or more GIPSY tier. Technically, a GIPSY node is a controller that wraps tier instances. Operationally, a GIPSY node hosts one *Tier Controller* for each kind of tier. The tier controller acts as a factory that will, when necessary, create instances of this tier. This model promotes the scalability of computation by allowing the creation of new tier instances when the existing nodes get overloaded or lost. It also provides the possibility for

the future implementation if any automated optimization mechanism come true to enhance the performance of GIPSY computing units.

GIPSY Instance

A GIPSY *instance* is a set of interconnected GIPSY Tiers deployed on GIPSY Nodes executing GIPSY programs by sharing their respective GEER instances. A GIPSY Instance can be executing across different GIPSY Nodes, and the same GIPSY Node may host GIPSY Tiers that are part of separate GIPSY Instances. In order to have a comprehensive understanding of "tier", "node", and "instance", we introduce a sample GIPSY network in Section 3.2.4.

3.2.3 Detailed Design

Context

In the sense of intensional programming, a context is a point of reference in the multidimensional context space in which an identifier is evaluated. For example, a program identifier might vary in the `time` dimension, such as in the following definition:

```
wealth = income
       - expenditures
       + previous wealth
```

where `wealth` varies in discrete time, or intervals of time, for example:

```
wealth@[time:july] = income@[time:july]
                   - expenditures@[time:july]
                   + wealth@[time:june]
```

The preceding notation `[time: january]`, refers to a mapping between a dimension (`time`) and a tag defined on this dimension (`january`). This kind of context is called context element. A context is defined as a set of context elements, where each element refers to a different dimension, such as: `[time: july, account: savings]`.

Demand in Detail

In our implementation of eduction, a demand is a request for the value of a program identifier, a Lucid identifier or a procedure identifier embedded in a hybrid Lucid program. Generally, demands have the following form:

(GEERID, PROGRAMID, *context*)

where GEERID is a unique identifier for the GEER that this demand was generated for; PROGRAMID is an identifier declared in this GEER (in the case of intensional demand, a Lucid identifier; in the case of procedural demand, a procedure identifier); and context is for the context of evaluation of the demand.

Types of Demands

Intensional Demand A demand for the evaluation of a Lucid identifier, given a certain context. Intensional demands are created and further processed by the Demand Generator Tier. It has the following form:

(GEERID, PROGRAMID, *context*)

Procedural Demand A demand for the evaluation of a certain procedure (originally written in a procedural language, as part of a hybrid GIPSY program or a method in an Object-Oriented Intensional Programming Language). Procedural Demand are generated by the Demand Generator and will be processed by the Demand Worker.

(GEERID, PROGRAMID, OBJECT PARAMS[], *context*, [CODE])

In procedural demands, OBJECT PARAMS[] is an array of objects that this procedure takes as arguments, and [CODE] is the optional code of the procedure.

Resource Demand A demand for a processing resource, i.e. a resource that is necessary for the evaluation of demands. Demand Generators will create resource demands for GEER instances if they receive demands to be computed for a GEERid that they are not aware of. Similarly, when a Demand Worker receives a demand for which it does not have the corresponding procedure class

to execute, it will create a Resource demand for that procedure class. A resource demand has the following form;

(RESOURCETYPEID, RESOURCEID)

where the RESOURCETYPEID is an identifier for a resource type, which is an enumerated type now containing GEER and `ProcedureClass`. This enumerated type is expandable in order to allow new resource types to be added later. The RESOURCEID is the unique identifier for the specific resource instance being sought for by the demander. Any new resource type created must be provided with a unique identifier scheme to identify each specific resource instance of this type.

Demand States

As shown in Figure 8, when a demand migrates from one tier to another, its state is changed in order to be properly identified for the next step.

Pending A *pending* demand is a demand that has been issued by a tier but not yet "grabbed" by any other tier for further processing. When a tier notifies its availability for processing demands, a query is made for a pending demand.

InProcess A *processing* demand is a demand that has been "grabbed" by a tier, and whose evaluation is still being processed, i.e. its result is not yet available. This state is assigned in order to make sure that the same demand is grabbed by only one tier for processing.

Computed A *computed* demand is a demand that is finished being computed, i.e. its "result" member is now assigned a value. Any time a demand is issued, a query is made to check for the presence of this demand in the Demand Store. If a computed demand is found, it is returned, thus saving its computation time.

Demand Generator Tier

The *Demand Generator Tier* (DGT) generates intensional demands and procedural demands according to the program declarations stored in the GEER generated for a GIPSY program. The intensional demands will be further processed by a DGT (either locally or by other DGT) and the

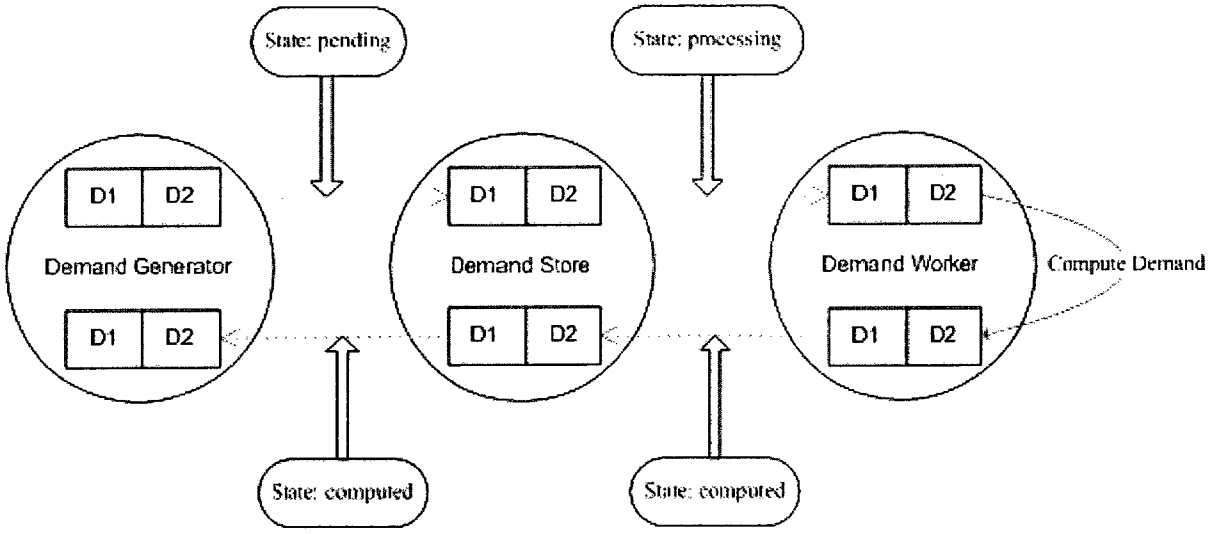

Figure 8: Transition of Demand States (procedural demand execution)

procedural demands will be dispatched through the Demand Store Tier to be further processed by the Demand Worker. *Local Demand Pool* is a pool of GEER instances that this DGT instance can process demands for. The *Local Demand Store* is an output buffer to allow the accumulation of generated demands.

Demand Worker Tier

The *Demand Worker Tier* (DWT) is a tier that can process procedural demands. If a DWT instance receives a demand for which it does not have the capacity to process, it can issue a Resource Demand for the required *Procedure Class*. Local GEER pool and Local Demand Store are also contained in DWT as well, beyond that, *Procedure Class* is an unique concept for DWT, procedure class is a Java class embedding a procedure that can be executed by the DWT. This procedure can be a Java procedure or a procedure written in any other languages that can be embedded into a Java class, e.g. using JNI to declare a "wrapper" class to embed this procedure so that it can be called by the Java Virtual Machine.

Demand Store Tier

The *Demand Store Tier* (DST) acts as a middleware to migrate demands between tiers, and also acts as a warehouse to store demands and their resulting values. One of the unique concept of DST

$$\mathcal{A} = \mathcal{B} + \mathcal{C};$$
$$\mathcal{B} = f(...) + \mathcal{D};$$
$$\mathcal{C} = 2;$$
$$\mathcal{D} = i@[time : july];$$

Figure 9: Set of Lucid Equations

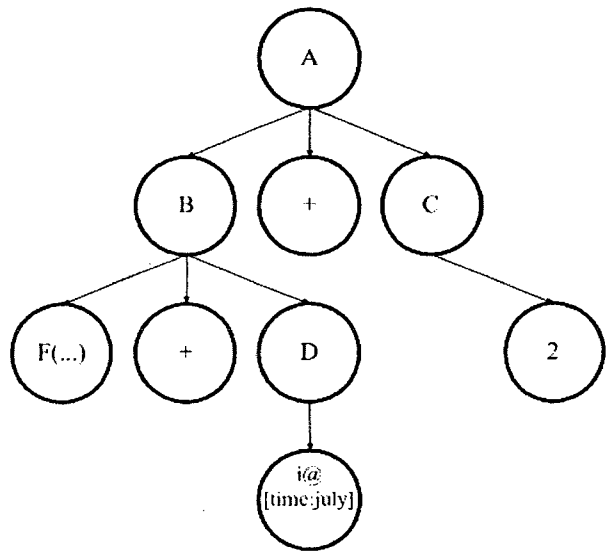

Figure 10: The Abstract Syntax Tree for Identifier $\mathcal{A}$

is the *Transport Agent*. Each DST instance exposes Transport Agents (TAs) to other tiers, which can use a lookup service to connect to the exposed TAs. How we integrate the existing Demand Migration System applications into our design will be presented in Section 3.3 and Section 3.4.

Workflow of the Demand Execution Process

In order to properly comprehend the entire process of demand execution, here we briefly explain all steps of demand migration. For this purpose, we provide a simple example to demonstrate the level of granularity a demand may be divided into.

In the presentation of the workflow, given the simplistic Lucid example in Figure 9, its Abstract Syntax Tree (AST) for identifier $\mathcal{A}$ is shown as Figure 10.

1. The engine initially generates a set of intensional demand and places them in DGT's Local

Demand Pool. For example, an initial demand might be:

$$(\text{GEER}1, \mathcal{A}, cxt1) \qquad (1).$$

2. The DGT picks up one demand in its Local Demand Pool, analyzes and find out which GEER it requires. The first element of the *Demand Signature* represents the requested GEER (e.g., GEER 1).

3. The DGT checks if it has the requested GEER (i.e., GEER1) in its Local GEER Pool.

4. As part of our basic assumption, DGT has the GEER (i.e., GEER1) in its Local GEER Pool by default, so it starts examining the related AST for the identifier (i.e., $\mathcal{A}$). An identifier may depend on the values of other identifiers or functions or computations. Therefore, the DGT should have all their results available in order to have the final value for that identifier (i.e., $\mathcal{A}$), in the context pertaining to the demand being processed.

5. After initial examination of the demanded value's AST in the corresponding GEER, i.e., as shown in Figure 10, each identifier may relate to some other identifiers, which means that each variable has some children nodes in its AST. Upon having those children nodes, new demands would be initiated according to the dependencies of the original identifier (i.e., $\mathcal{A}$). For example, as shown in Figure 10, identifier $\mathcal{A}$ relates to both values of $\mathcal{B}$ and $\mathcal{C}$. Thereafter, it generates two demands for the result of each of them, and put them into the Local Demand Pool as the following demands, and puts the initial demand (1) on hold until these demands are computed:

$$(\text{GEER}1, \mathcal{B}, cxt1) \qquad (2)$$
$$(\text{GEER}1, \mathcal{C}, cxt1) \qquad (3)$$

6. In computing demand (2), it is analyzed that $\mathcal{B}$ itself depends on both function $f()$ and $\mathcal{D}$. Thus, the DGT generates new demands at this point, presented below as (4), (5), and further (6). Among those demands, $\mathcal{C}$ being a constant, it has its immediate value hard-coded in the AST without any additional computation.

$$(\text{GEER}1, f2, cxt1, (...)) \qquad (4)$$

$$(\text{GEER1}, \mathcal{D}, cxt1) \qquad (5)$$

$$(\text{GEER1}, i, (cxt1 + [time : july])) \qquad (6)$$

7. As all these demands are being generated, the Local Demand Pool sends a remote request to the DST to see if these demands (i.e. intensional demand (6) for identifier i) have already been computed by other GIPSY nodes and stored in the store. In the affirmative, the computed values are retrieved and may be used directly to follow up on the computation. In the negative, these demands might be computed locally or sent to the DST be caught by another DGT and processed.

8. Procedural demands, (e.g., demand 4), are similarly conveyed to the DST, with the difference that procedural demands are picked up by a DWT for processing.

9. For now, we assume that we have all required GEER installed in the Local GEER Pool of DGT and DWT, so it can process demand without any problem.

10. After finishing the computation of a demand, a DGT or DWT sends the result back to the DST. The DST then stores the result in its demand pool and will notify all DGTs which make request for this demand that the value is available.

3.2.4 Deployment View

Figure 11 [Paq09] is a comprehensive description of the concept of "tier", "node" and "instance". In Figure 11 there are 6 GIPSY Nodes (identified by bounding rectangles and named as Node 1..6), Node 1 is a self-contained GIPSY instance that does not use tiers running on remote nodes. In this case, the GIPSY node and GIPSY instance share the same set of tiers. Note that this GIPSY instance has a DGT and a DWT instance, thus it can execute hybrid programs. A GIPSY instance not including any DWT instance could only compute pure Lucid programs, i.e. Lucid programs that do not include calls to procedural functions. In fact, any GIPSY instance must include at least one DST instance and at least one DGT instance. The only tier that can be absent is the DWT, and only in cases where a GIPSY instance executing pure Lucid programs.

Node 2 is composed of a DGT instance and a DST instance. For example, the DGT Instance might have been assigned the execution of a certain GEER. The demands it generates during the

execution of this GEER are sent to its local DST instance, using the TA_1 that the latter exposes internally to Node 2 tier instances. As this DST Instance is connected to other DST Instance in GIPSY Instance II using its exposed TA_2 and TA_3, the demands can be grabbed by other DWT or DGT instances in this GIPSY instance. In the case that these remote DWT or DGT Instances do not have the GEER corresponding to these demands, the remote tiers can issue a resource demand for this GEER, which will eventually be responded to by the DGT instance in Node 2, after which the remote TAs will be able to install this GEER in their GEER Pool and process these demands.

As is the case for Node 2, all tiers in Node 3 are allocated to GIPSY instance II. However, this node includes a set of DWT instances, and is thus unable to generate demands locally, but can only respond to procedural demands issued by remote DGT instances (in Node 2 and Node 4) and store their computed values in its local DST instance.

Interestingly, Node 4 does not include any DST instance, and hosts many DWT instances and a single DGT instance. These all necessitate a remote TA to be exposed by a remote node in order to connect to a GIPSY instance. Node 4 also has three DWT instances that are not yet connected to any TA. Using their TA lookup procedure will eventually get them the list of remotely exposed , and then connect to one of them.

As the same theory, the GIPSY tiers in black that resident in node 5 and 6 form the third GIPSY instance.

3.3 The Demand Migration System

As mentioned in Section 3.2.1, we divided the multi-tier system into three sub-modules and the DST takes charge of migrating demands and archiving values. In order to increase the flexibility of the application, two DMS applications had been implemented with Jini and JMS in [Vas05] and [Pou08]. These two DMSs follow the same framework, but are not 100% isomorphic, in order to integrate them into the multi-tier architecture, we still have to perform certain refactoring work.

In Section 3.3.1 we introduce the concept of middleware and which principles the two DMSs are supposed to follow, in Section 3.3.2, Section 3.3.3, and Section 3.3.4 we describe the Jini and JMS implementation of DMS, in the end (Section 3.3.5) we make a brief comparison between them.

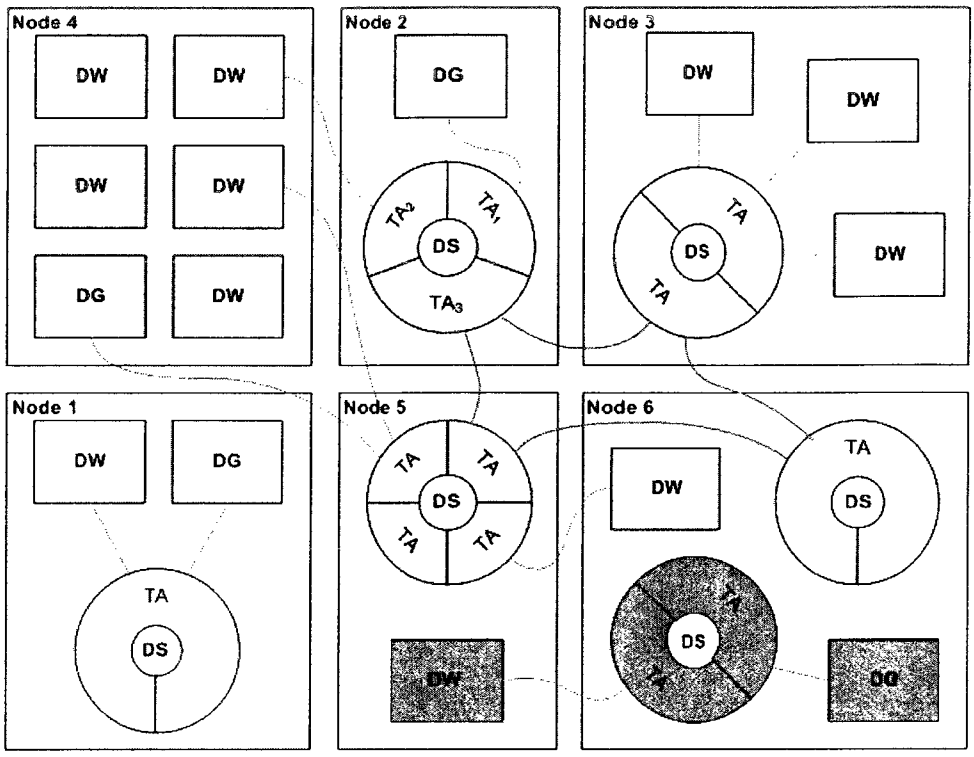

Figure 11: Deployment View of GIPSY Network

3.3.1 Middleware

Middleware is the software positioned between the operation system and the application. As mentioned in the textbook distributed systems architecture [PRP06]:

> *Viewed abstractly, middleware can be envisaged as a 'tablecloth' that spreads itself over a heterogeneous network, concealing the complexity of the underlying technology from the application being run on it.*

In the runtime system of GIPSY, the Demand Migration System (DMS) falls in the concept of middleware to take the role of demand store tier (DST) which has been described in Section 3.2.3. As a middleware, we expect the DMS follow such principles:

1. A single natural object-oriented design for a given application, regardless of the context in which that application will be deployed;

2. Failure and performance issues are tied to the implementation of the components of an application, and consideration of these issues should be left out of an initial design;

3. The interface of an object is independent of the context in which that object is used.

As a matter of fact, the DMSs are still in need of interface integration and internal parameter unification. Later this section we will briefly introduce the existing Demand Migration System and in Section 3.4 we will discuss what are the necessary steps to make them work as the middleware that we expect.

3.3.2 Demand Migration System as a Framework

Demand Migration System (DMS) is a generic framework for migrating objects in a heterogeneous and distributed environment, particularly, migrate demands among GIPSY execution nodes. Now two DMS applications based on the distributed technologies of JINI [Vas05] and JMS [Pou08] have been created, in which a layered structure (Figure 5) is applied. The DMS consist of two principal functional layers: *Demand Dispatcher Layer* and *Demand Migration Layer.*

3.3.3 JINI-DMS

JINI-DMS incorporates a solution based on JINI and JavaSpace [Fle01], where JINI has been used for the design and implementation of the Transport Agents and JavaSpace for the design and implementation of the Store. The JINI-DMS was the first DMS instance, developed by Emil Vassev in his Master thesis [Vas05].

JINI features and Resource Components

JINI, developed by Sun Microsystems is an infrastructure for federating services in a distributed system and also provides an open architecture for handling resource components. The resource components are handled as services and the JINI system provides mechanisms for their construction, lookup, and communication. JINI is a pure Java technology and integrates easily with the GIPSY which is entirely implemented in Java. Access to many of the services in JINI system is lease based. A lease represents the time of validity of a particular entry. If a lease if not refreshed, it will expire, and consequently, the entry is deleted from the registry.

JavaSpace

JavaSpace, as a part of the JINI framework,is a network accessible associated share memory to share, exchange and store Java objects. It hides the internal details of persistence, distribution from developers while leaving them free to build distributed data driven applications.

3.3.4 JMS-DMS

JMS-DMS incorporates a solution based on Java Message Service [Sun07], Jboss [JBo07] and Hypersonic Database (HSQLDB) [The10], where JMS is a set of interfaces and associated semantics that govern the access to messaging systems, Jboss as the JMS provider is a messaging system that implements the JMS interfaces and provides administrative and controlling features, and the Hypersonic Database is an embedded solution inside Jboss Application Server kit to provide persistency and caching. The JMS-DMS was the second DMS instance, developed by Amir Hossein Pouteymour in his Master thesis [Pou08].

Java Message Service

Java Message Service (JMS) is a set of Java API allowing the applications to create, send, receive and read the messages. As illustrated in Figure 12 [Pou08], the JMS application consists of two parts, the applications itself and the JMS Stub. When it comes to tasks that require any remote computation, it uses JMS Stub to communicate with the JMS server and receive the messaging service. JMS specification does not define how the server should be implemented, but rather defines the interfaces and services that the JMS infrastructure must provide.

JMS Provider

JMS Provider acting as the central part, leverages all administrative, functional, and control capabilities of the JMS messaging. Among the many commercial and open-source providers, JBoss Application Server [JBo07] has been selected, a comprehensive study is provided in [Pou08].

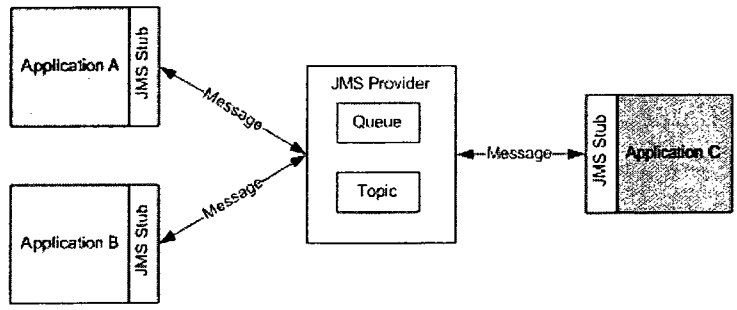

Figure 12: Basic JMS Messaging Architecture

3.3.5 JINI-DMS vs JMS-DMS

Except using different technologies, although the two DMS instances intended to follow the same framework, variations in the APIs and implantation make the demand migration framework fail to accord with the concept of framework instances. For example, the JINI Demand Dispatcher communicated directly with the JavaSpace, and was called by `JINITransprotAgentProxy`. The `JMSTA` communicated with Message Queue directly and had no `DemandDispatcher`. And also, the client of `IDemandDispatcher` and `ITransportAgent` were not compatible to each other which means they need different implementations of the DWT, etc. We are not going to discuss too much detail here, that will be covered in Chapter 4, but rather to declare that refactoring is necessary for the DMSs in order to make the multi-tier architecture generic.

3.4 Methodology

From Section 3.2 and Section 3.3 we know that the main focus of this thesis breaks into two parts, a general framework of Multi-Tier runtime system and wrapping the DMSs into the framework.

GIPSY being an ongoing research project since 2001, it has a development context that is different from commercial or open-source software. First, since it's a research project, it is open and full of possibilities, the design and implementation may change or even head to other directions with the evolution of research without strictly following the software requirement specifications. Second, most of the programmers involved are students, each one may only take care of one or several sub-modules of the whole system during his or her academic years, the code may or may

not be documented properly. Given the above reasons, with the growing of the code base, software maintenance and refactoring becomes extremely hard. As we know, refactoring is risky. It requires changes to working code that may introduce subtle bugs. As Martin Fowler stated in the book [FBB+00]:

> *You start digging in the code. Soon you discover new opportunities for change, and you dig deeper. The more you dig, the more stuff you turn up...and the more changes you make. Eventually you dig yourself into a hole you can't get out of. To avoid digging your own grave, refactoring must be done systematically.*

In order to carry out the implementation and refactoring *systematically*, we borrowed some ideas from *Agile Development*, and adopt a subset of *Extreme Programming* practices. In Appendix A we present a brief discussion about agile development and how we apply the methodology in the research.

3.5 Summary

In this chapter, we described the design of the Multi-Tier Architecture. We introduced several key concepts of GIPSY which affect the design of the runtime system at the beginning in Section 3.1. In Section 3.2 we draw the blueprint of the runtime system and explained the design rationale. Section 3.3 is a description of previous developed demand migration system which will be integrated. At the end, in Section 3.4, we briefly described the methodology used for the implementation. In the next chapter, we will provide detailed information about the implementation.

Chapter 4

Implementation & Experiments

"The designer of a new kind of system must participate fully in the implementation."

Donald E. Knuth

In the last chapter we have drawn a blueprint of the multi-tier architecture, in this chapter we describe how the implementation is performed. Since much of the work involves refactoring the existing components and collaboration with other ongoing modules, at the beginning, in Section 4.1 we explain our testing infrastructure, in Section 4.2 we discuss the implementation of the multi-tier runtime framework which is a set of wrappers that covers other separated components, in Section 4.3 we introduce how we carry out the refactoring and integrate the Demand Migration System into the multi-tier architecture. As mentioned in Section 3.4, we borrowed ideas from agile development, especially test-driven development, the unit testing will be presented with the evolving of the code in each section. In Section 4.4 we integrate the multi-tier runtime system into the GIPSY code base and show the executing result.

4.1 Testing Approach

4.1.1 Test-Driven Development

As mentioned in Section 3.4, for the development of Mulit-Tier Runtime System, we decided to apply *test-driven development* (TDD), which is a software development technique that relies on the repetition of a very short development cycle: First the developer writes a failing automated test

case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

In GIPSY, there are three main modules: the General Intensional Programming Compiler (GIPL); the General Eduction Engine (GEE), and the Run-time Interactive Programing Environment (RIPE). Accordingly we now have 3 main test packages which are `tests.GIPC`, `tests.GEE` and `tests.RIPE`. We follow this infrastructure to put testing for Multi-Tier Architecture under the `gipsy.tests.junit.GEE.multitier` package.

4.1.2 Unit Testing

For each class we have built, thought should be given to how the class is to be tested. As we have seen, the extreme programming (in Section A.2) approach suggests that a test set should be created before any coding starts. This is not as simple as it seems because at the start of the coding of a unit, it may not be easily defined. So what is important here is that a basic framework for testing the unit is defined, and this will be developed into a more detailed set of testing in tandem with the coding. At the end of the initial exploratory coding stage, a complete set of tests should then be available so that thorough testing of the class is possible.

Given the outline description or structure of a class, we have to identify two important things: First, what are the ways in which the method will be accessed and what, if any, are the preconditions on the data that is supplied to it? Second, what are the ranges of values that need to be provided for the methods?

Once we have identified these, the expected outputs have to be considered and, in particular, action taken to ensure that the output information of interest can be read or display in an appropriate form. JUnit, which is a unit testing framework for the Java programming language, has been used to write test cases used in conjunction with the tested class code to establish whether it is behaving in a desired manner.

The test suites have to provide the information needed to prepare the class for testing, and this involves identifying the entry points to the method and supplying suitable data to make the test work. In any method that we want to test, there will be some data input values needed from a defined data structure or type. It is important to ensure that the data selected for this purpose is sufficiently varied to expose the method to all possible types of failure as well as success. We are

trying to do two things during testing: gain some confidence that the method works and at the same time try to break it. Only then can we be sure that the class is trustworthy enough to be considered for integration into GISPY.

4.1.3 Integration Testing

As their name implies, unit tests are concentrating on the localized aspects of individual units. However, in order to test if he units are working properly together as a system, units need to be connected and tested together. Such testing is called *Integration Testing*. For example in our case we have to test system-level situations where tiers are started, connected and communicate across a network. In order to achieve that, we have to setup proper configuration objects prior to execution, start each tier, have them use the lookup service to connect together, etc. Such tests can only be done when several units are put in conjunction and operate together, which goes beyond the scope of unit testing.

The main focus point of this thesis is the propagation of demand in the multi-tier architecture. In the integration testing, we setup three scenarios to migrate and process the *intensional*, *procedural*, and *resource* demands. And in each integration test, we inspect the status of each demand at different processing point in the workflow to make sure that the implementation is really working according to our workflow design.

In Section 4.4 we present the technical description of our integration testing after we integrate the demand migration system into multi-tier runtime system and as a whole running in the GIPSY framework.

4.2 The Multi-Tier Runtime System

In this section we present the conceptual design and some of the implementation details of the multi-tier architecture's realisation. It is important to know that in order to integrate the multi-tier framework into GIPSY we have to depend on some of the features that are already provided by GIPSY and also have to reuse some of the existing components. In Section 4.3 we describe the refactoring that has been performed on the previously developed demand migration system and in Section 4.4 we introduce the integration of our resulting work in the existing code base.

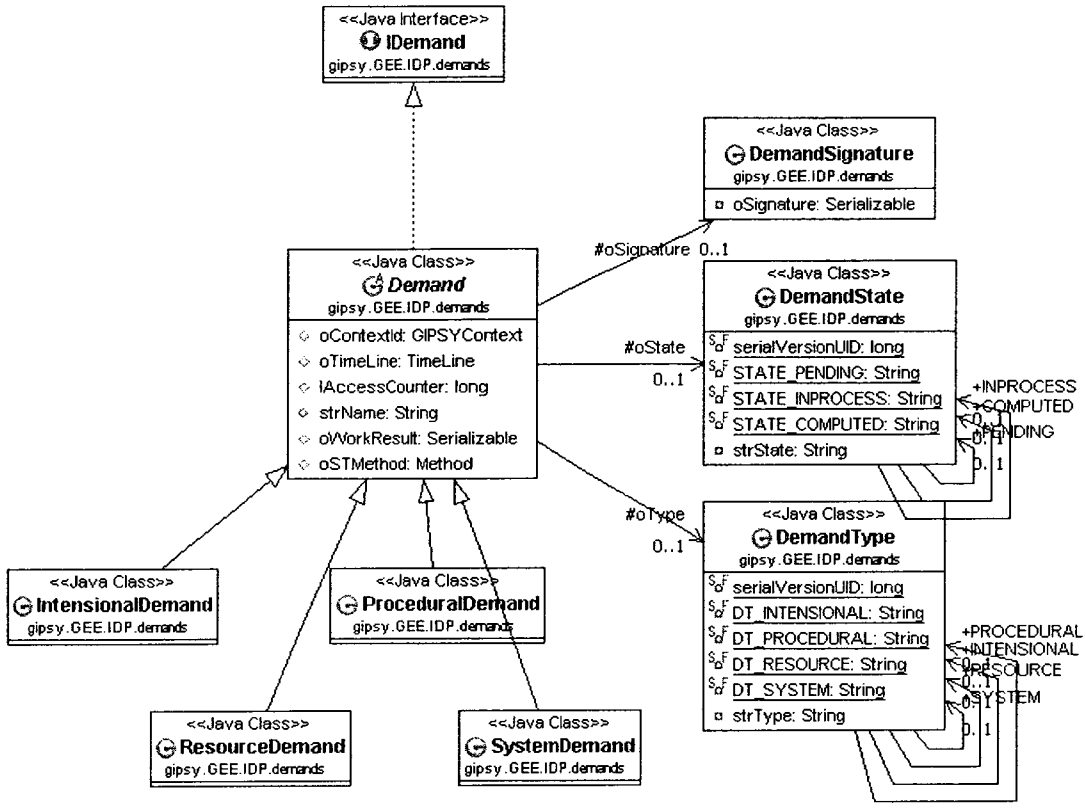

Figure 13: Demand Class Diagram

4.2.1 Demands

As stated conceptually in Section 3.2.3, the execution model of the GIPSY run-time system relies on demand-driven evaluation. The value of Lucid variables depend on the context of their evaluation. A demand is conceptually a identifier-context pair, and yields a result. So the the demand is the main data structure migrated in the runtime system. Each Demand has its own type, state and signature. The class diagram for the Demand class is shown in Figure 13.

Demand Signature The class `DemandSignature` encapsulates a serializable universal identifier that provides the unique identifier for each demand.

Demand State The class `DemandState` represents an enumeration type used to distinguish the demands by their state. Three states are defined - *pending*, *in process*, and *computed* and implements functions for determining the state of each demand. Listing 4.1 depicts the implementation.

```
public class DemandState
implements Serializable
{
        private static final String STATE_PENDING = "pending";
        private static final String STATE_INPROCESS = "inprocess";
        private static final String STATE_COMPUTED = "computed";
        ...
        public boolean isPending(){}
        public boolean isInProcess(){}
        public boolean isComputed(){}
        ...
}
```

Listing 4.1: Demand State implementation

Demand Type As demand state, `DemandType` is also an enumeration to encapsulate the type of demand. It defines three type for now - *intensional demand procedural demand*, and *resource demand.*

In Section 4.4.1, we use the `DemandFactory` to generate each kind of demand, and process them in the integration testing.

4.2.2 General Implementation Architecture

4.2.2.1 Multi-Tier Package

Classes and interfaces for the implementation under the `gipsy.GEE.multitier` package. The corresponding wrappers are located in their respective subdirectories (sub-packages). To summarize, we have a root multitier package:

`gipsy.GEE.multitier`

which is separated into sub-packages for each of the tier types and the corresponding wrapper classes among other things:

`gipsy.GEE.multitier.DGT`

`gipsy.GEE.multitier.DST`

`gipsy.GEE.multitier.DWT`

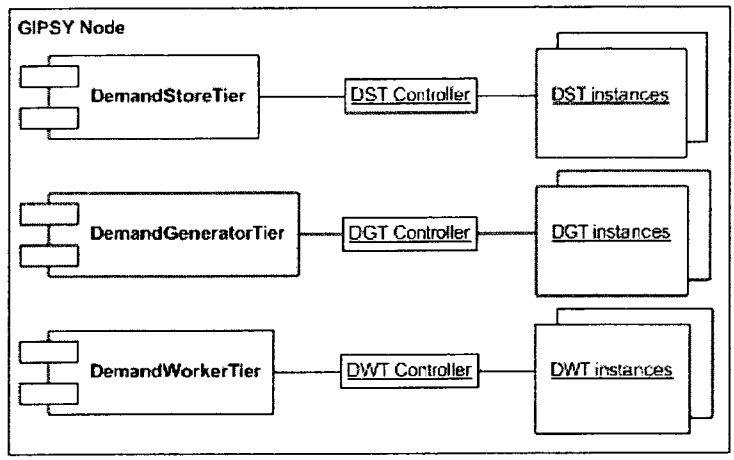

Figure 14: Design of the GIPSY Node

4.2.2.2 GIPSY Node

Abstractly, a GIPSY Node is a computer that has registered for the hosting of one or more GIPSY tier. Technically, a GIPSY Node is a controller that wraps GIPSY Tier instances, and that is remotely reporting. Operationally, a GIPSY Node hosts one tier Controller for each kind of tier (see Figure 14). The Controller acts as a factory that will, when necessary, create instances of the tier, which provide the concrete operational features of the tier in question. This model permits scalability of computation by allowing the creation of new tiers instances as existing tier instances get overloaded or lost. Figure 15 is the class diagram of node controller implementation.

In the implementation, we design an abstract class `TierController` which is the base class for all the three controllers. Three specific classes inherit from this one, `DGTController`, `DSTController`, and `DWTController`. As we all know when we make use the `new()` method of class `TierController`, code may have to be changed as new concrete classes are added, in other words, the code will not be "*closed for modification*", to extend it with new concrete types, we will have to reopen it. So in order to identify the aspects that vary and separate them from what stays the same, we decided to apply *Simple Factory Pattern*, (see Figure 15).

We take the `TierController` creation code and move it out into another object that is only going to be concerned with creating controllers, the `ControllerFactory`. The factory handles the details of object creation. Once we have `ControllerFactory`, the `GIPSYNode` just becomes a client of it. Any time it needs a controller it asks the factory to make one. Now only the

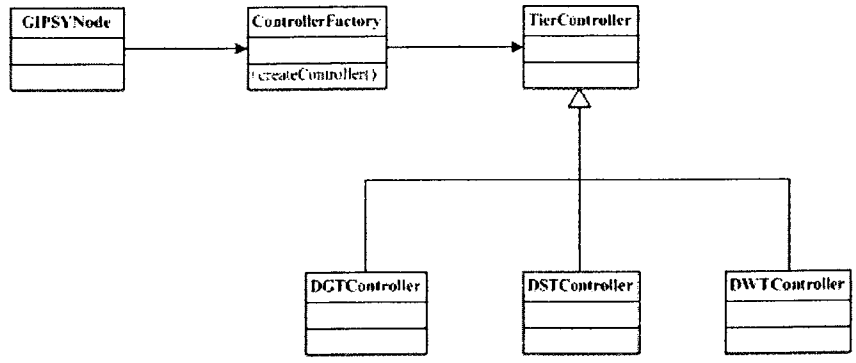

Figure 15: Class Diagram of GIPSY Node Controller

`ControllerFactory` takes care of controller creation. Listing 4.2 is the pseudo code for the creation of controllers.

```
public class ControllerFactory{
        ...
        public TierController createController(type)
        {
                TierController controller = null;

                if (type.equals(DGT))
                {
                        controller = new DGTController();
                }
                if (type.equals(DST))
                {
                        controller = new DSTController();
                }
                if (type.equals(DWT))
                {
                        controller = new DWTController();
                }

                return controller;
        }
        ...
}
```

Listing 4.2: Implementation of createController Method

4.2.2.3 Wrappers API and Classes

Regarding the core design of the Multi-Tier Architecture and the developers' implementation efficiency, we decided to define the three aforementioned wrapper classes: `DGTWrapper` (Demand Generator Tier Wrapper), `DSTWrapper` (Demand Store Tier Wrapper), and `DWTWrapper` (Demand Worker Tier Wrapper) such that they all inherit from the same abstract class called `GenericTierWrapper`

that implements the most common functionality of the interface `IMultiTierWrapper`. We defined the API of the interface, and we provide the code for it and adjust the tier wrapper stubs to adhere to the interface.

The interface we designed is placed into the same package `gipsy.GEE.multitier` and is called `IMultiTierWrapper`. The initial content of the interface is in Listing 4.3 (trimmed) that the above actual wrapper classes implement. Thus, we define the actual Java syntax interface and its implementation within the concrete wrapper classes.

```
import gipsy.Configuration;

public interface IMultiTierWrapper
extends Runnable
{
    ...
    startTier();
    stopTier();
    setConfiguration(Configuration);
    Configuration getConfiguration();
    ...
}
```

Listing 4.3: Primary API of the `IMultiTierWrapper` Interface.

4.2.2.4 Generic Tier Wrapper

Two GIPSY objects included as data members in the `GenericTierWrapper` (see Figure 16) adhere to the APIs of `Configuration` and `ITransportAgent`, and are described further. All wrappers are to have a configuration instance and potentially communicate through a given transport agent (TA) implementation.

`Configuration` contains a `Serializable` configuration of this GIPSY instance and its components, for static and run-time configuration management. As in Listing 4.4, the key concept of the `Configration` class is the `oConfigurationSettings` data member that is an object of `java.util.Properties`, the `Properties` class inherits from `java.util.Hashtable`, represents a persistent set of properties.

The `Properties` class represents a persistent set of properties. The properties can be saved to a stream or loaded from a stream, for example we can save it as xml files. Each key and its corresponding value in the property list is a string. Since the `Properties` object might be accessed by different methods at the same time, we *synchronized* the related operation to prevent concurrency

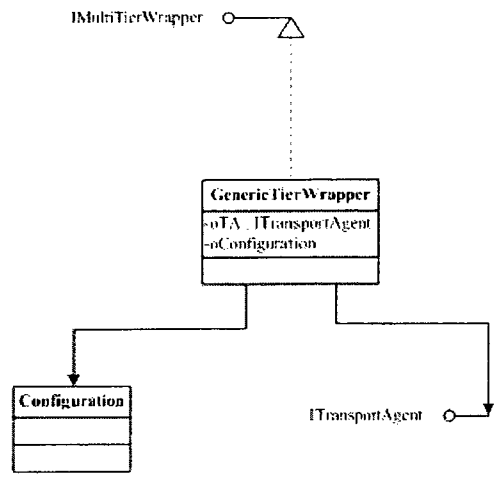

Figure 16: `GenericTierWrapper` Class Diagram

problems. And also the `Properties` might be exchanged or transformed over a network, it needs to be `Serializable`.

```
import gipsy.Configuration;

public class Configuration
implements Serializable{
        ...
        protected Properties oConfigurationSettings = null;
        ...
        public String getProperty(String pstrKey, String pstrDefaultValue)
        public String getProperty(String pstrKey)
        public synchronized Object setProperty(String poKey, String poValue)
        public synchronized Object setProperty(GIPSYContext poContext)
        ...
        public synchronized void load(InputStream poIn) throws IOException
        public synchronized void loadFromXML(InputStream poIn) throws IOException,
            InvalidPropertiesFormatException
        public synchronized void store(OutputStream poOut, String pstrComments) throws
            IOException
        public synchronized void storeToXML(OutputStream poOut, String pstrComment, String
            pstrEncoding) throws IOException
        ...
}
```

Listing 4.4: Coding Configuration Class

A TA reference is abstracted by the `ITransportAgent` unification interface for all transport agents (TAs) implemented in the extended DMF (Demand Migration Framework) and DMS (Demand Migration System). For the sake of homogeneity, all TAs must implement this interface. This is a super-interface for the use by the different tiers in the multi-tier architecture. The original implementations based on Jini and JMS did not have a common super-interface, which we

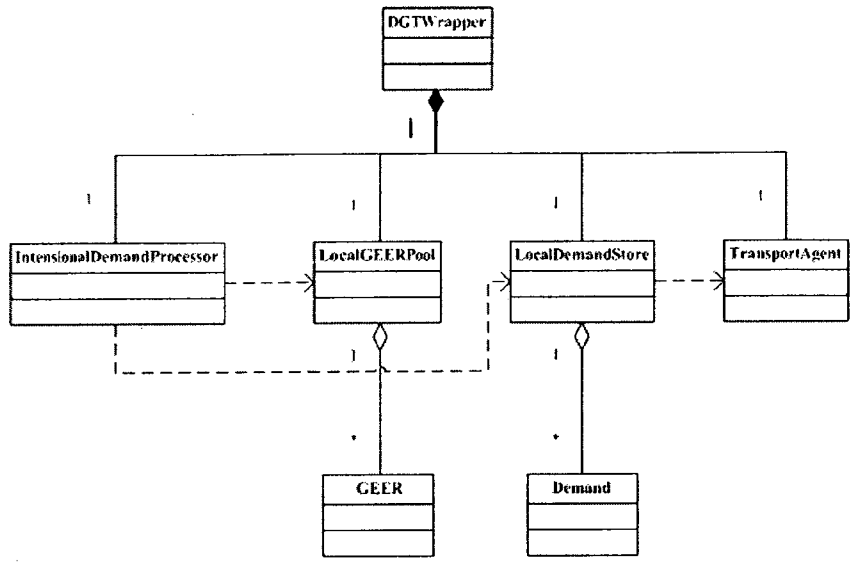

Figure 17: Demand Generator Tier Wrapper Class Diagram

had to define and provide ourselves during the course of this work. After defining a family of interfaces, we can encapsulate each implementation and make them interchangeable. The *strategy pattern* lets the implementation technique vary independently from clients that use it. More detailed implementation of `GenericTierWrapper` will be covered in Section 4.2.6.

4.2.3 Demand Generator Tier Wrapper

The Demand Generator Tier generates intensional demands and procedural demands according to the program declarations stored in the GEER generated for a particular GIPSY program. Some of the demands generated locally will be dispatched through the Demand Store Tier to be further processed by other generators or workers. Figure 17 is the class diagram for Demand Generator Tier Wrapper.

The Demand Generator Tier uses a Local GEER Pool that contains the GEER instances that this Demand Generator Tier instance can process demands for. A GEER instance is a dictionary that contains definitions for each identifier in a GIPSY program. Each Lucid identifier is linked to an abstract syntax tree. The Intensional Demand Processor traverses this tree in order to generate further demands, according the eduction model of computation. Note that if a Demand Generator Tier receives a demand to be processed for a GEER that is not in its Local GEER Pool, it may

send a resource demand for it. Such demands for GEER instances are propagated and stored as resource demands in Demand Store Tiers, thus achieving the possibility of executing several GIPSY programs concurrently and seamlessly, potentially using several Demand Generator Tiers that can share GEER instances on demand.

Given an initial intensional demand for the value of a certain identifier in a certain context, the Intensional Demand Processor "extracts" further demands from the Lucid program declaration, as stored as an AST in the GEER, by traversing the AST and generating further demands accordingly. The Intensional Demand Processor then sends them to its Local Demand Store, which acts as a buffer for outgoing demands. When the Intensional Demand Processor is finished with the generation of demands related to one particular initial demand, it will look into its Local Demand Store for a demand still in need of further processing, and will start generating demands for it.

The Local Demand Store is an output buffer to allow the accumulation of processed demands, in case of unavailability of the Transport Agent while the results of demands currently being processed are created by the Procedural Demand Processor. Interestingly, in cases where a simple pure Lucid program is to be executed in a lean runtime environment, the Local Demand Store can be used as a local demand store by not registering to a Transport Agent.

Implementation

Figure 18 is the sequence diagram for DGT. Considering both Local GEER Pool and Local Demand Store are working as a buffer to store Demand and Demand Signature pairs, it is important to handle insert and query operations efficiently, thus, they are implemented with `java.util.HashMap` and wrap the default methods.

Unit Testing

When the new implementation was going to be introduced, unit testing should be addressed to make sure the new piece of code is working according to the workflow involved with the DGT. Figure 19 is the class diagram of test cases for `DGTWrapper`, as well as is internal components `LocalGEERPool`, and `LocalDemandStore`. All these tests (16 in total) have proven to pass successfully.

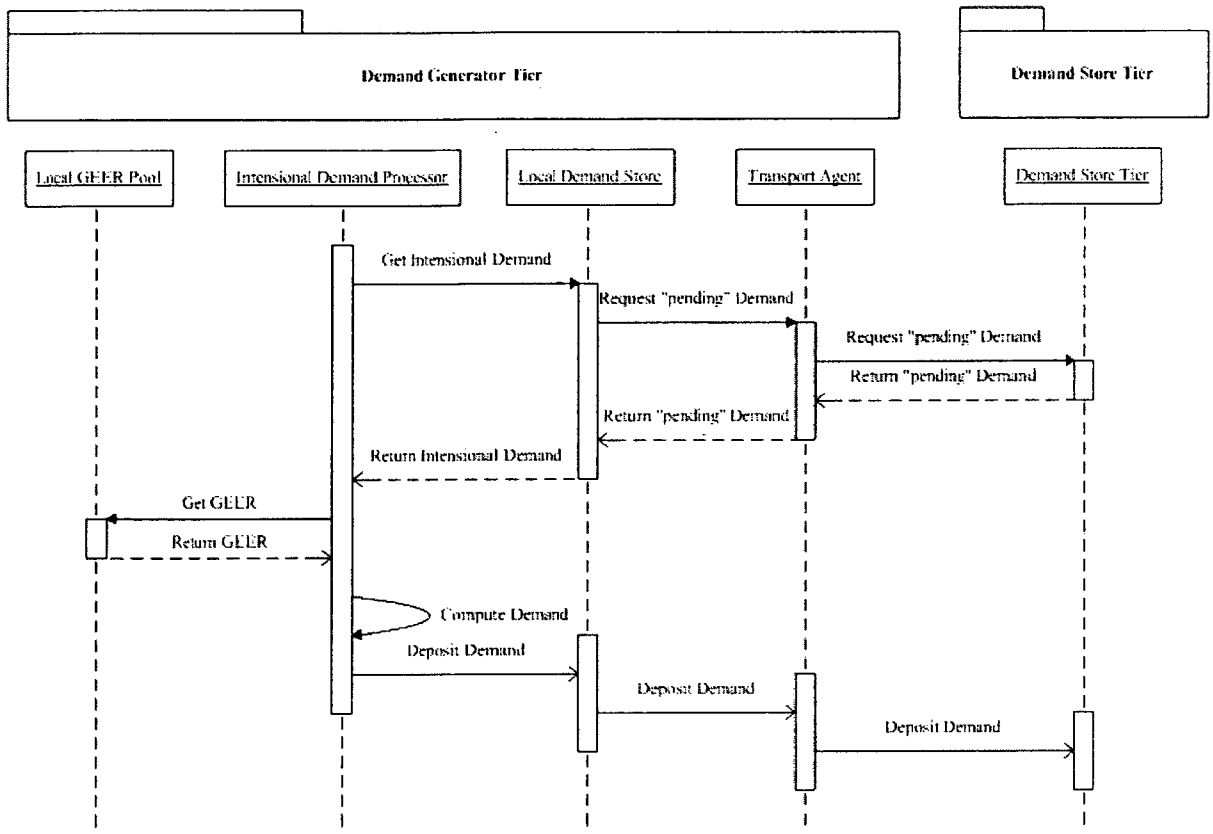

Figure 18: Demand Generator Tier Sequence Diagram

4.2.4 Demand Store Tier Wrapper

The Demand Store Tier (DST) (see Figure 20) acts as a middleware between tiers in order to migrate demands between them. In addition to the migration of the demands and values across different tiers, the Demand Store Tier provides persistent storage of demands and their resulting values, thus achieving better processing performances by not having to re-compute the value of every demand every time it is eventually re-generated after having been processed. From this latter perspective, it is equivalent to the historical notion of *warehouse* in the eduction model of computation. The Demand Store Tier is implemented using JMS/Jini as part of the Demand Migration Framework (DMF) [VP05a].

The DST uses Dispatcher Entry objects wrapping a demand for the purpose of migration using the Demand Migration Framework (DMF). Dispatcher Entries are created when the Demand Dispatcher uses a Transport Agent to dispatch a demand. The DGT in fact stores Dispatcher

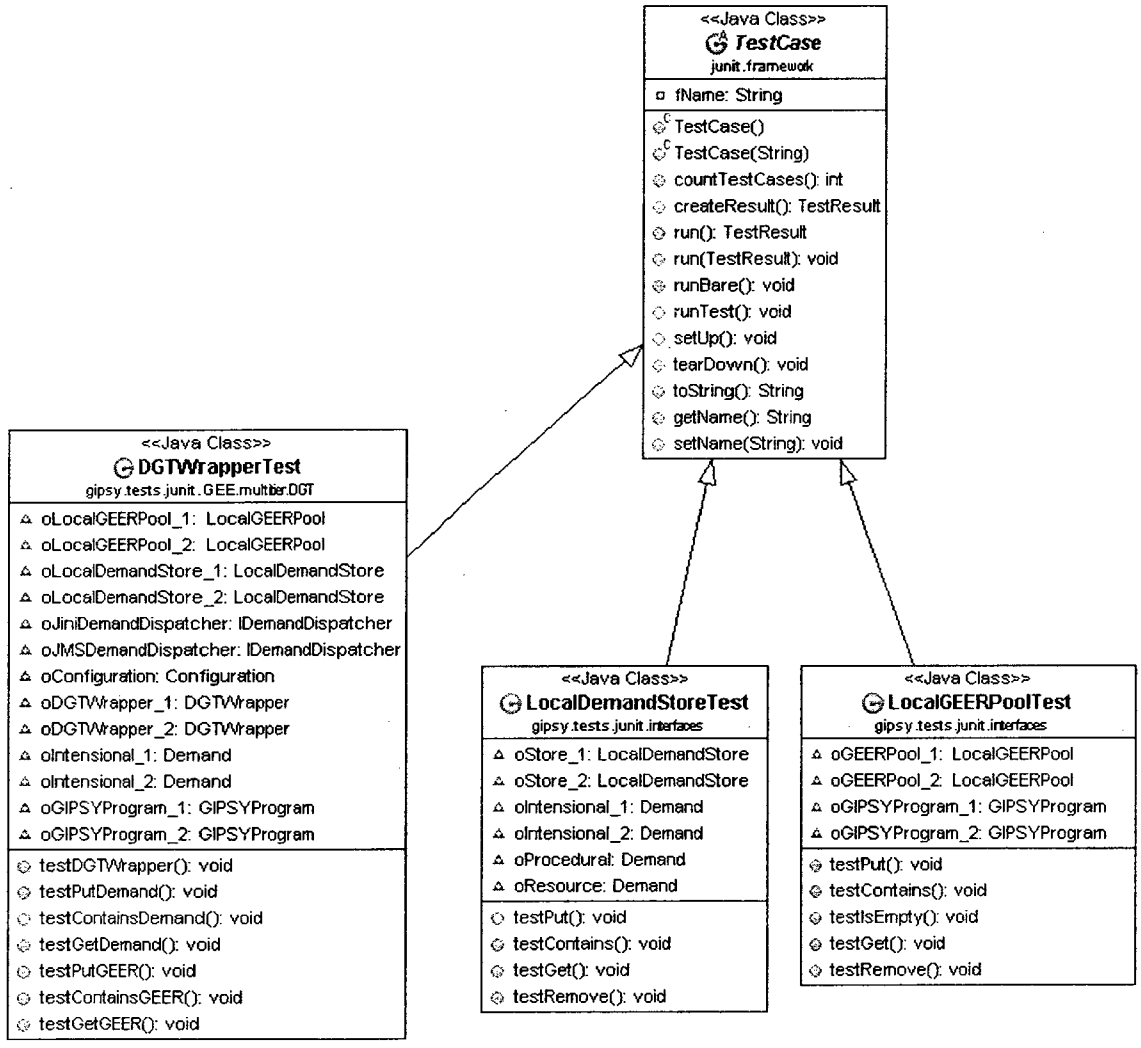

Figure 19: Test Case for `DGTWrapper`, `LocalGEERPool` and `LocalDemandStore`

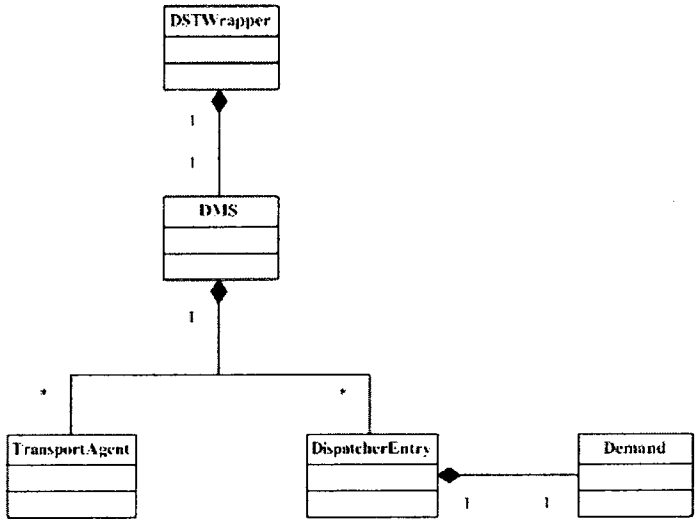

Figure 20: Design of the Demand Store Tier

Entries that encapsulates an individual demand and exposes an interface usable by the DMF. Only the DMF views the demands as Dispatcher Entries. Each DST Instance exposes Transport Agents (TAs) to other tiers, which can use a lookup service to connect to the exposed TAs.

Implementation

According to the previous implementation, *JavaSpace* is used as a part of the JINI framework, which is a network accessible associated share memory to share, exchange and store Java objects. It hides the internal details of persistence, distribution from developers while leaving them free to build distributed data driven applications. JMS-DMS incorporates a solution based on Java Message Service [Sun07], Jboss [JBo07] and Hypersonic Database (HSQLDB) [The10], the Hypersonic Database is an embedded solution inside Jboss Application Server kit to provide persistency and caching. Thus, JavaSpace and Jboss Application Server (HSQLDB) work as the DST to holding the computed and pending demands.

Unit Testing

Apart from having been refactored, the Demand Store Tier is essentially untouched by our development. Each of the previously developed classes for the implementation of the demand migration systems have been properly tested. Hence, we do not provide additional unit tests for the DST.

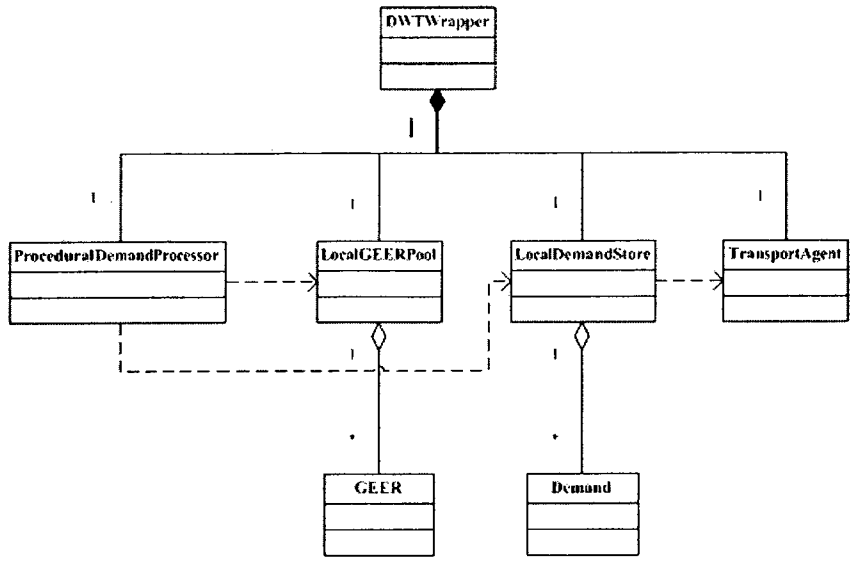

Figure 21: Demand Worker Tier Wrapper Class Diagram

However, the DST is subject to integration testing as described later in the chapter.

4.2.5 Demand Worker Tier Wrapper

The Demand Worker Tier is a tier that can process procedural demands. It consists of a Procedural Demand Processor that can process the value of any procedural demand corresponding to one of the elements of its Procedure Class Pool, which represents executable code for all the procedural demands that this Demand Worker Tier instance is able to respond to. The Procedure Class Pool is embedded in the Local GEER Pool. Figure 21 is the class diagram for Demand Worker Tier Wrapper.

The Procedural Demand Processor represents the "worker" part of this tier. It receives pending procedural demands from its associated Transport Agent, identifies what Procedure Class this demand refers to by the use of the demand signature and `GEERid` of the demand, which is a search key in its Local GEER Pool. In order for the Worker to optimize its processing time, all procedural demands are wrapped in threaded classes.

Similarly to the DGT, the DWT uses a Local Demand Store as an output buffer to allow the accumulation of processed demands, in case of malfunction of the Transport Agent while the results of demands currently being processed are created by the Procedural Demand Processor. It also

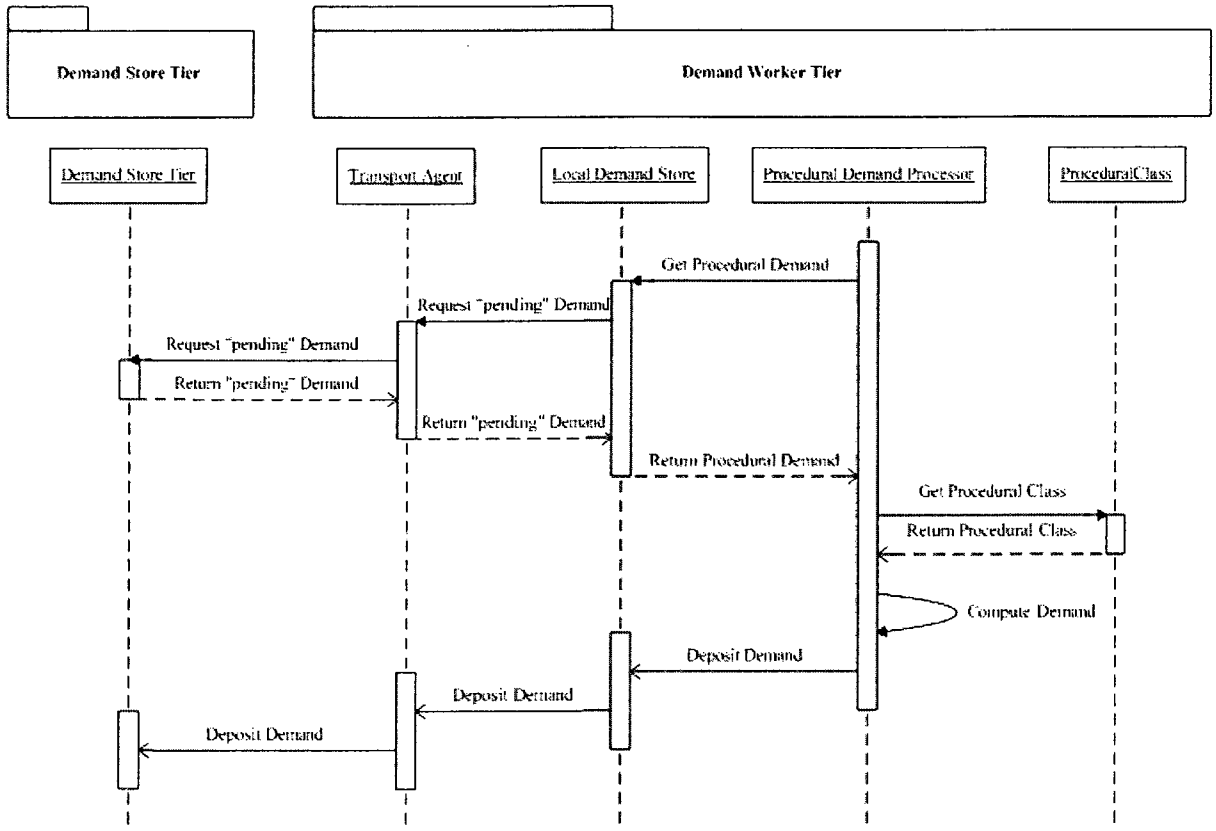

Figure 22: Demand Worker Tier Sequence Diagram

uses the notion of Local GEER Pool, but in this case the DWT is concerned with the Procedure Classes embedded in the GEER. The procedure classes are generated by the GIPC, and included into the GEER at compile time.

Working Scenario

Figure 22 is the sequence diagram for DWT

4.2.6 GenericTierWrapper Implementation

The `GenericTierWrapper` is the abstract class that implements `IMultiTierWrapper` interface and inherited by `DGTWrapper`, `DSTWrapper`, and `DWTWrapper`, from the discussion in Section 4.2.3 and

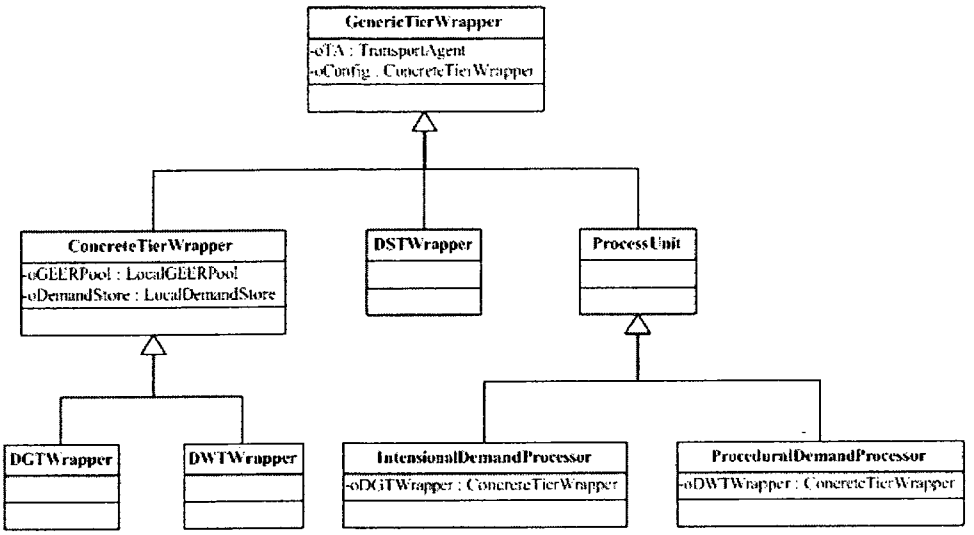

Figure 23: Multi-Tier Wrapper Class Diagram

Section 4.2.5, considering their composition in Figure 17, Figure 21 and working manner in Figure 18, Figure 22, we can make the conclusion that the DGT and DWT are isomorphic, the only difference are the processing units, `IntensionalDemandProcessor` and `ProceduralDemandProcessor` that more look like the decorator of DGT and DWT. And also because the Intensional Demand Processor is more attached to the original Generic Eduction Engine (see Section 2.4), GIPSY group have decided to separate the implementation of Intensional Demand Processor with the Multi-Tier Architecture, so that will not be covered in this thesis, but we do have to take that component into consideration, there are two reasons: the first one, the extensibility of the software, as part of the design work, we need to make sure the Multi-Tier Runtime system to be easily extended to incorporate new behavior without modifying the existing code, the principle that is *Classes be open for extension, but closed for modification*; the second reason, in order to deliver the integration testing, we still have to reserve the Intensional Demand Processor component. As stated above, we decided to apply *Decorator Pattern* to the design and implementation of `GenericTierWrapper` and its subclasses. Figure 23 shows the class diagram of redefined implementation.

The following advantages are provided by applying the decorator design pattern to develop the `IntensionalDemandProcessor` and `ProceduralDemandProcessor`:

1. Ease further implementation of the `IntensionalDemandProcessor`, which uses a simulator for the present integration testing.

2. Open for any new subclasses of `GenericTierWrapper` when needed.

3. The behavior comes in through the composition of decorators with the base components as well as other decorators, we can develop new behaviors at any time to add new behavior without changing the existing code.

4.3 Refactoring of the Demand Migration Systems

In this section, we discuss the refactoring that has been delivered on the previous implemented DMSs.

4.3.1 Demand Migration System

To overcome GLU's inflexibility, the GIPSY was designed to include a generic and technology-independent collection of Demand Migration Systems (DMSs), which implement the Demand Migration Framework (DMF) for a particular technology or technologies to communicate and store information. Jini-DMS [Vas05] incorporates a solution based on Jini [Jin07] and JavaSpaces [Mam05, Fle01], where Jini has been used for the design and implementation of the Transport Agents (TAs) and JavaSpaces for the design and implementation of the Demand Store.

JMS-DMS [Pou08] applied the DMF framework based on the Java Messaging Service (JMS) paradigm. The JBoss Application Server [JBo07] has been used as JMS provider and Hypersonic Database (HSQLDB) [The10], which is an embedded solution inside JBoss, provides persistence and caching.

As a generic framework for migrating objects in a heterogeneous and distributed environment, particularly, migrate demands among GIPSY execution nodes, DMS consist of two principal functional layers: *Demand Dispatcher Layer* and *Demand Migration Layer.*

Demand Dispatcher Layer

In general, the Demand Dispatcher Layer maintains a pool of demands to be processed. The following elements describe the Demand Dispatcher's internal structure.

Dispatcher Proxy (DP) The Dispatcher Proxy inherits the Presentation Layer (PL), it works as a proxy for the Demand Dispatcher. The Demand Dispatcher relies on it to expose functionality to its clients. The clients are Transport Agents, Demand Generators and Demand Workers. All the Demand Dispatcher's clients are assigned with a unique Demand Proxy. The clients use the Demand Proxy functions as their own, in their local address space, thus hiding the complexity of a possible remote collaboration with the Demand Space.

Demand Space (DS) The Demand Dispatcher relies on the Demand Space to store all the pending demands and their computed results. The Demand Space layer implies all the characteristics of an Object Database, i.e., the Demand Space provides a mechanism to store the state of objects persistently, and an Object Query Language (OQL) to retrieve these objects. The Object Database Management Group (ODMG) published this standard in 1993 [CB+00].

Demand Migration Layer

The Demand Migration Layer establishes a context for migrating objects among the GIPSY tiers and nodes. The Migration Layer provides a transparent form of migration. The Migration Layer refers to communication between computers and its architecture is based on the Open Systems Interconnection (OSI) Reference Model [Vas05, Day95]. In addition, the Migration Layer provides an architectural structure, forming a multi-platform transport protocol that is able to connect machines with different operating systems. The Migration Layer focuses on the use of Transport Agents, which are special kind of autonomous messengers.

Transport Agent (TA) Transport Agents are based on distributed technologies whose architecture influences their implementation. Transport Agents differ in their structure and implementation, but they all expose the same interface to the Demand Generators, Demand Workers and Demand Dispatcher. Thus, despite the distributed technology diversity, their services are transparent and homogeneous with regard to their API.

TA Interface When a Transport Agent starts, it plugs into the system by connecting with the Demand Dispatcher and exposes its interface to Demand Generator and Demand Worker instances. Actually, the DWs and DGs listen constantly to DST for newly plugged Transport Agents and when

the latter appear, they connect to them. Thus, DWs and DGs must adhere to the Transport Agent interface in order to connect to that Transport Agent.

4.3.2 Jini DMS

JINI-DMS incorporates a solution based on JINI and JavaSpace [Fle01], where JINI has been used for the design and implementation of the Transport Agents and JavaSpace for the design and implementation of the Store. The JINI-DMS was the first DMS instance, developed by Emil Vassev in his Master thesis [Vas05].

4.3.3 JMS DMS

JMS-DMS incorporates a solution based on Java Message Service [Sun07], Jboss [JBo07] and Hypersonic Database (HSQLDB) [The10], where JMS is a set of interfaces and associated semantics that govern the access to messaging systems, Jboss as the JMS provider is a messaging system that implements the JMS interfaces and provides administrative and controlling features, and the Hypersonic Database is an embedded solution inside Jboss Application Server kit to provide persistency and caching. The JMS-DMS was the second DMS instance, developed by Amir Hossein Pouteymour in his Master thesis [Pou08].

4.3.4 Refactoring Tasks

Following the analysis above, there are some aspects of the existing DMS that have been refactored to adjust to suit the Multi-Tier design, the refactoring was delivered with Serguei A. Mokhov, Yi Ji and the author. In order to maintain the integity of the development process, we present them here as a whole:

- The `JiniTransportAgent` and `JMSTransportAgent` don't share the same interface. The original implementations based on Jini and JMS did not have a common super-interface, which we had to define and provide ourselves during the course of this work. After defining a family of interfaces, we can encapsulate each implementation and make then interchangeable. The strategy pattern lets the implementation technique vary independently from clients that

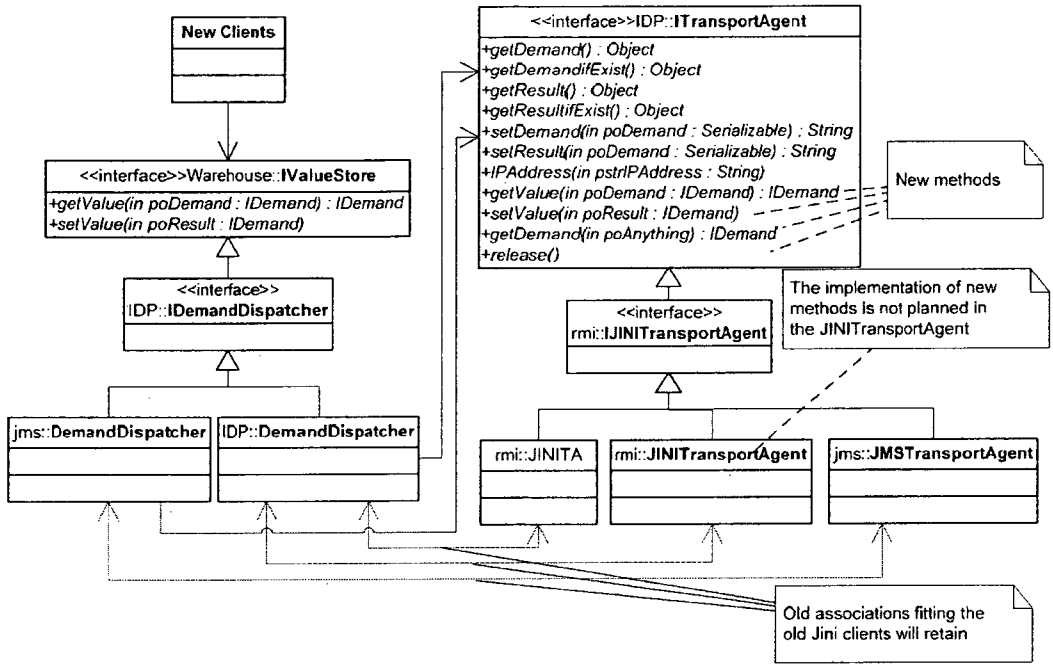

Figure 24: DMS Refactoring Completion Class Diagram

use it. `ITransportAgent` as the super interface, all the Transport Agent implementations, including `JiniTransportAgent` and `JMSTransportAgent` inherit from it.

- The `Demand` is not used in both DMSs, the data being migrated in the system need to be unified to our `Demand` interface, and also introduce the concept of `DemandSignature`.

- The `JINIDemandDispatcher` communicates directly with the JavaSpace, and was called by `JINITransprotAgentProxy`, but The `JMSTransportAgent` communicates with Message Queue directly and had no `DemandDispatcher` involved. In order to make sure the system is technique independent, we made the `IDemandDispatcher` directly inherit the `IValueStore` interface. The Transport Agent handles the communication between itself and the Demand Store directly, and called by Demand Dispatchers. Thus, the client of the Demand Dispatcher and the Demand Dispatcher itself don't need to be aware of the `DispatcherEntry`.

The following phases have been delivered for the refactoring:

1. In order to make the Jini and JMS DMS implementation work in the same manner, we add a `JMSTransportAgent` inheriting the `IJINITransportAgent` and `JMSDemandDispatcher`.

2. Create the `IValueStore` interface, define its corresponding new methods and let the `IDemandDispatcher` directly inherit from it. Figure 24 is the class diagram upon refactoring completion.

4.3.5 Integration of DMS into Multi-Tier Architecture

After the above refactoring, the following interfaces Listing 4.5 and Listing 4.6 are publicly posted, and ready to use, both Jini and JMS DMS implement these two interface.

Now, the Demand Dispatcher can be viewed as a store access object, each `DGTWrapper` possesses a reference of `IDemandDispatcher`, thus, the `DSTWrapper` is transparent to the `DGTWrapper`, each generator only need to invoke the `writeDemand()` and `readDemand()` methods to deposit the demand into the store and withdraw the result from it after the demand is computed by the worker.

Impact of Refactoring on the `DGTWrapper`

For the `DGTWrapper`, to cooperate with DST, we added two inner classes: `DepositDemand` and `WithdrawResult`, both of them are inherited from GIPSY's `BaseThread`.

`DepositDemand` is the thread to deposit pending demand into the DST through the Demand Dispatcher, specifically, the pending procedural demand is send to the DST and waiting to be computed by the DWT, and the pending intensional demand is processed locally or remotely by the DGT, then send the result to the DST.

`WithdrawResult` is the thread to withdraw the result from the DST, particularly, `WithdrawResult` queries the *inprocess* demand archived in the local demand store from DGT, once it's in the DST and computed, the result will be send to DGT.

Impact of Refactoring on the `DWTWrapper`

Within `DWTWrapper`, the same theory as `DGTWrapper` applied, we created two inner classes: `DepositResult` and `WithdrawDemand` (different from `DepositDemand` and `WithdrawResult`), these two threads run when the `DWTWrapper` start.

```
public interface IDemandDispatcher
{
        /**
         * Write demand into a store.
         */
        public DemandSignature writeDemand(IDemand poDemand)
        throws DemandDispatcherException;

        /**
         * Write a result into the store.
         */
        public DemandSignature writeResult(DemandSignature poSignature, IDemand poResult)
        throws DemandDispatcherException;

        /**
         * Read a task from the store.
         */
        public IDemand readDemand()
        throws DemandDispatcherException;

        /**
         * Read a task, if exists, from the store.
         */
        public IDemand readDemandIfExists()
        throws DemandDispatcherException;

        /**
         * Read a result from the store. This method blocks the client's
         * thread but is used in a different way from {@link #getValue(IDemand)}
         */
        public IDemand readResult(DemandSignature poSignature)
        throws DemandDispatcherException;

        /**
         * Read a result, if exists, from the store.
         */
        public IDemand readResultIfExists(DemandSignature poSignature)
        throws DemandDispatcherException;

        /**
         * Cancel a demand already dispatched to the store.
         * If the demand is already proceeded, then the result will be canceled.
         */
        public void cancelDemand(DemandSignature poSignature)
        throws DemandDispatcherException;

        /**
         * Get the value of the specified demand and block.
         */
        public IDemand getValue(IDemand poDemand)
        throws DemandDispatcherException;
}
```

Listing 4.5: IDemandDispatcher Interface

```
public interface ITransportAgent
extends Serializable
{
        /**
         * Get the pending demand from the store.
         * This is usually a blocking method.
         */
        IDemand getDemand()
        throws DMSException;

        /**
         * Get any pending demand if it exists.
         * This is a non-blocking method and may return null.
         */
        IDemand getDemandIfExists()
        throws DMSException;

        /**
         * Get the result of the demand with the specified signature
         * from the store. Once called, this method would wait until
         * there is a result or an exception emerges.
         */
        IDemand getResult(DemandSignature poSignature)
        throws DMSException;

        /**
         * Peek to see if there is a result for the specified demand,
         * and return it if there is any. This method does not block
         * the caller's thread.
         */
        IDemand getResultIfExists(DemandSignature poSignature)
        throws DMSException;

        /**
         * Puts the demand into the store.
         */
        DemandSignature setDemand(IDemand poDemand)
        throws DMSException;

        /**
         * Puts the result back into the store
         */
        DemandSignature setResult(IDemand poResult)
        throws DMSException;

        /**
         * Client IP address setting.
         */
        void setClientIPAddress(String pstrIPAddress);
}
```

Listing 4.6: ITransportAgent Interface

`WithdrawDemand` is the thread to withdraw pending demand from DST through the transport agent, the pending demand, most of time the pending procedural demand are executed by the DWT, then store the result, set state to computed, and archived temporarily in its local demand store.

`DepositResult` is the thread to deposit the computed demand that have been cached in the local demand store and using the transport agent to put back in the DST.

4.4 Integration Testing

In the previous section we have discussed how we integrated the DMS into multi-tier runtime system. In this section we are introducing how we run the multi-tier system in the global scope of the whole GIPSY system. It is important to note that, since the heterogeneity, capacity, and effectiveness of the DMS was tested in [Vas05], the quality of service and comparison of Jini and JMS technology were conducted in [Pou08], the ultimate goal of our integration testing which is also the purpose of this thesis is that: the two demand migration applications have been wrapped into the multi-tier workflow, and all type of demands are propagated and executed properly.

4.4.1 Eduction Engine Simulator

In this section we introduce the notion of demand generator simulator in the context of our integration testing strategy. There are three reasons why we chose to rely on a simulator rather than the regular demand generator:

1. In the multi-tier architecture, the DGT processes demands according to a GEER and generates demands according to it. The implementation of the eduction engine and its demand generator is currently under redesign and its current implementation is not compatible with our own design.

2. For integration testing purposes, we would like to test that the migration of the demand is following our designed workflow. So it is important that the generator is under as much control as possible and generates the demand (demand type, state) that we are looking for. This is easier to achieve with a simulator.

3. The purpose of this thesis is the demand migration but not the demand generation which is mainly addressed by the GIPC and the GEE.

We have named our demand generator simulator the `DemandFactory`. The most important methods of the simulator are listed as follow Listing 4.7:

```
public class TestDemandFactory
{
        public IDemand create(DemandType poType) throws GIPSYException
        {
                if(poType.isIntensional())
                {
                        return createIntensionalDemand();
                }
                if(poType.isProcedural())
                {
                        return createProceduralDemand();
                }
                if(poType.isResource())
                {
                        return createResourceDemand();
                }
                return null;
        }

        public IntensionalDemand createIntensionalDemand()
        {...}

        public ProceduralDemand createProceduralDemand() throws DMSException
        {...}

        public ResourceDemand createResourceDemand()
        {...}
}
```

Listing 4.7: `DemandFactory` Class

Methods `createIntensionalDemand`, `createProceduralDemand`, and `createResourceDemand` create the respective demand types. Specifically:

1. The `createIntensionalDemand` generates a demand from a compiled Lucid program, following a request for the value of an intensional variable in a specific GEER. This is exactly following the external interface of an eduction engine generating an intensional demand.

2. The `createProceduralDemand` randomly generates one procedural demand from a collection of pre-set procedures that are assumed to be in the Procedure Class Pool of the workers that will grab them. This is exactly following the external interface of an eduction engine generating a procedural demand.

3. The `createResourceDemand` generates demand for certain GEER by its GEER identifier. This is exactly following the external interface of an eduction engine generating a resource demand for a GEER.

In each integration test, we process each of the different kinds of demand. At the beginning of each integration test, we give the workflow to be followed, and validate each integration testing by tracing the status of each demand and the content in the local demand store of each tier at different points in the workflow. Specifically the purpose of the integration testing are:

1. The generation and migration of three types of demand.
2. In intensional demand testing, we test the demand migration between DGT and DST, and the working of local demand store.
3. In procedural demand testing, we test migration between DGT, DST and DWT.
4. In resource demand testing, we test migration between DGT, DST and DWT, and also the working of local GEER pool.

Since this is the first time that separated sub-modules in GIPSY are connected (as shown in Figure 1), there are still certain limitations as mentioned in Section 5.2. Through the integration testing, we make sure that the whole system is working as a whole and follow our proposed workflow.

4.4.2 Intensional Demand Processing Testing

In this testing, we fill the local demand store with intensional demands, assuming that the required GEERs are stored in the local GEER pool. The testing for resource demand is addressed in the next test case. Figure 25 describes the scenario for the first test case.

4.4.2.1 Configuration

In order to start the test case, the following configuration steps need to be followed to establish proper system setup:

- On GIPSYNode 1:

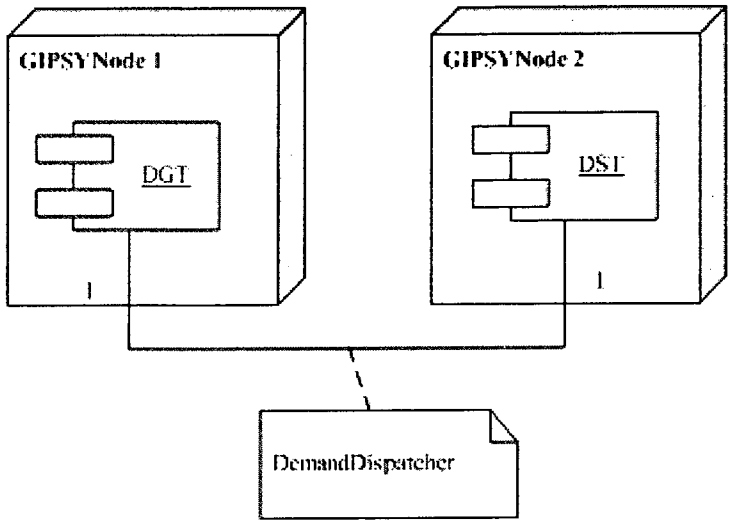

Figure 25: Intensional Demand Test Case

1. Run one DGT.
2. The DGT generates two intensional demands (pre-archived in the local demand store).
3. Run one `JiniDemandDispatcher` that connects to the DST, the `Configuration` object will read its values from the configuration file which is set to Listing 4.8.

- On GIPSYNode 2:

1. Run one DST - JavaSpace, for detailed information refer to Appendix D about how to setup Jini & JMS running environment.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="java.security.policy">bin/gipsy/GEE/IDP/config/jini.policy</entry>
<entry key="gipsy.jini.host">132.205.99.86</entry>
</properties>
```

Listing 4.8: Configuration File for Jini Demand Dispatcher

4.4.2.2 Workflow

The processing workflow (refer to Figure 26) should be:

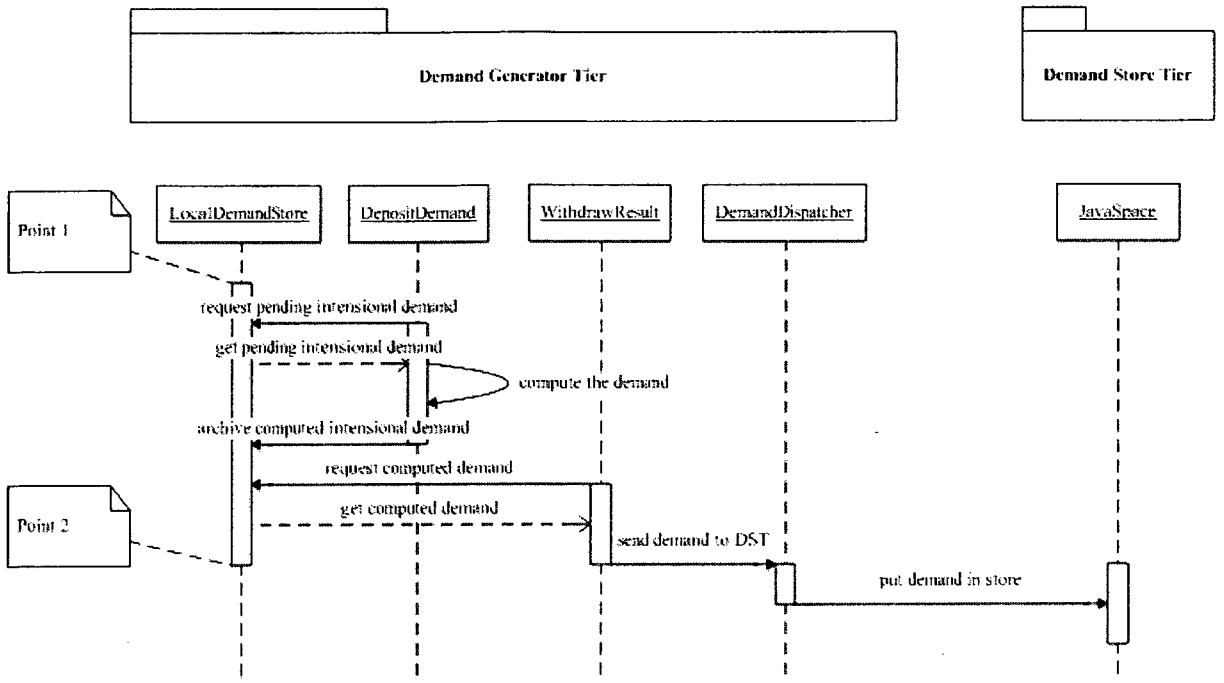

Figure 26: Intensional Demand Processing

1. `DepositDemand` retrieves a demand from `LocalDemandStore` and decides it should be passed to the DST or processed locally. Since Intensional Demand are executed by the DGT, then the demand is passed to the `IntensionalDemandProcessor`.

2. In order to compute the result, the DGT has to make sure the GEER that the intensional demand belongs to is in the `LocalGEERPool` (for this case, yes).

3. Computes the result, and passes it back to the `LocalDemandStore` and then pass the result to the DST.

4.4.2.3 Testing Result

1. The demands in the local demand store at Point 1 in Figure 26:

Before Computation		
Demand Type	Demand Signature	Demand State
INTENSIONAL	Dce8bb8f5-a06e-421c-8bde-c12cba68da5b	PENDING
INTENSIONAL	f77ca12f-0752-43a3-b09a-4c5b9e20e00e	PENDING

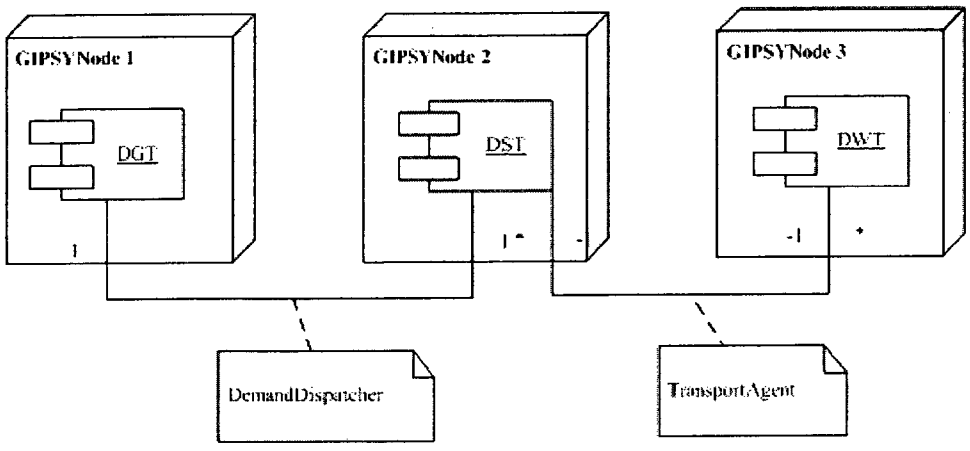

Figure 27: Procedural Demand Test Case

2. The demands in the local demand store at Point 2 in Figure 26:

After Computation			
Demand Type	Demand Signature	Demand State	Demand Result
INTENSIONAL	Dce8bb8f5-a06e-421c-8bde-c12cba68da5b	COMPUTED	int: 123
INTENSIONAL	f77ca12f-0752-43a3-b09a-4c5b9e20e00e	COMPUTED	int: 123

From the result we can see that the demand has been properly computed and migrated, and kept in the local demand store.

4.4.3 Procedural Demand Processing Testing

Figure 27 depicts the testing scenario, in this test case, we filled the local demand store of the DGT with procedural demands, the pending demands are sent to the DST, get grabbed and executed by the DWT, then the result gets back from the DST to the DGT.

4.4.3.1 Configuration

- On GIPSYNode 1:

  1. Run the DGT.

  2. The DGT generates two procedural demands (pre-stored in the local demand store).

3. Setup the Jini environment the same as Listing 4.8.

- On GIPSYNode 2:

  1. Run JavaSpace for DST.

- On GIPSYNode 3:

  1. Run DWT using a Jini Transport Agent, the same way setup Jini environment on GIPSYNode 1.

4.4.3.2 Workflow

The workflow for procedural demand processing (refer to Figure 28) should be as follows:

1. The `DepositDemand` of DGT retrieves a demand from the `LocalDemandStore`, since it is a procedural demand, it sets its state to inprocess, and passes it to the DST.

2. The `DepositResult` of DWT gets the pending demand from the DST, executes it, sets its state to computed, and then caches it in the local demand store of DWT, and then continues to fetch other demands.

3. The `WithdrawDemand` traverses the local demand store of the DWT, the computed demands are sent to the DST trough the transport agent, and then are removed from the store.

4. The `WithdrawResult` gets the result from the DST by its demand signature, then puts it back in the `LocalDemandStore` of the DGT.

4.4.3.3 Testing Result

1. The demands in the local demand store of DGT at Point 1 in Figure 28.

Before Computation		
Demand Type	Demand Signature	Demand State
PROCEDURAL	5e92f656-b569-4e8d-ae93-0866f90ee771	PENDING
PROCEDURAL	a3a69a58-0b35-4d9b-8032-abe8ae7cf5d2	PENDING

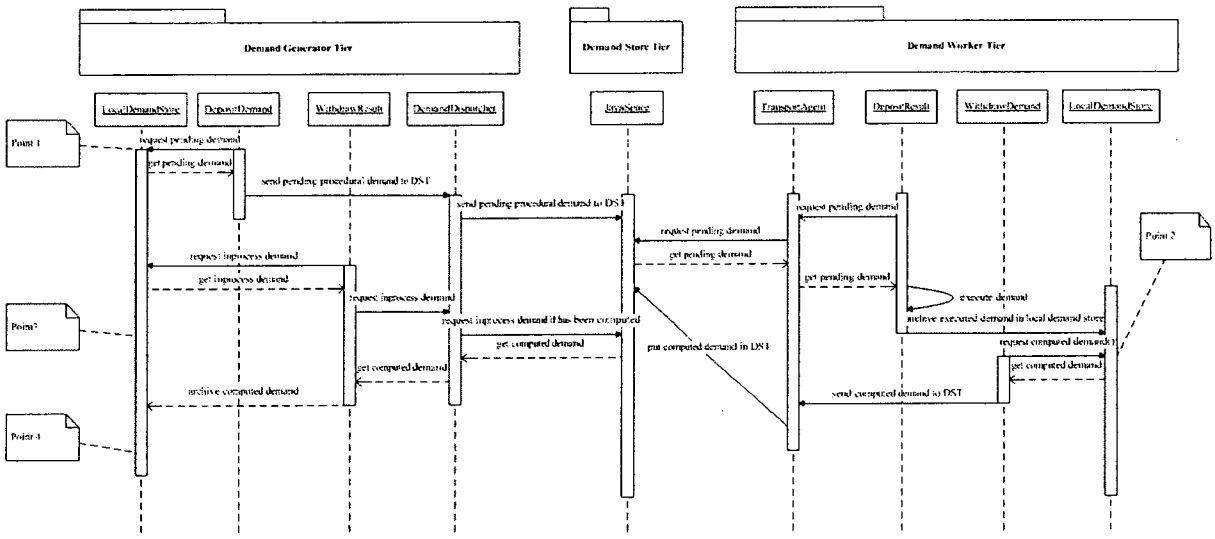

Figure 28: Procedural Demand Processing

2. The demand in the local demand store of DWT at Point 2.

Demand Type	Demand Signature	Demand State	Demand Result
PROCEDURAL	5e92f656-b569-4e8d-ae93-0866f90ee771	COMPUTED	[]

3. The demands in the local demand store of DGT at Point 3.

Demand Type	Demand Signature	Demand State
PROCEDURAL	5e92f656-b569-4e8d-ae93-0866f90ee771	INPROCESS
PROCEDURAL	a3a69a58-0b35-4d9b-8032-abe8ae7cf5d2	PENDING

4. The final states of the demands in the local demand store of DGT at Point 4.

After Computation			
Demand Type	Demand Signature	Demand State	Demand Result
PROCEDURAL	a3a69a58-0b35-4d9b-8032-abe8ae7cf5d2	COMPUTED	[]^1
PROCEDURAL	5e92f656-b569-4e8d-ae93-0866f90ee771	COMPUTED	[]^2

[1] Refer to Appendix E.1 for the result of this intensional demand.
[2] Refer to Appendix E.2 for the result of this intensional demand.

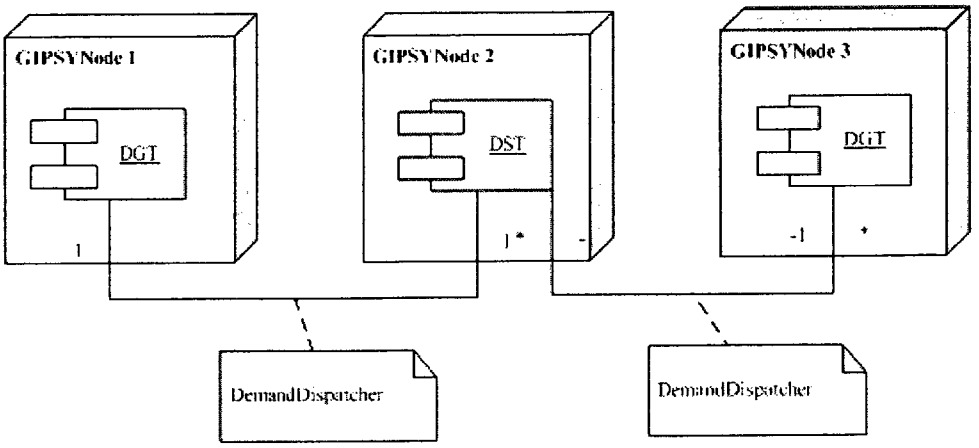

Figure 29: Resource Demand Test Case

4.4.4 Resource Demand Processing Testing

The `ResourceDemand` may ask for execution resources, in the context of this testing, it requests a `GEER`. Figure 29 depicts the testing scenario. At the beginning, we filled two resource demand in GIPSYNode 1, both of them request for a specific `GEER`, one (0.8474930470440531) is contained in DGT's `LocalDemandStore` of GIPSYNode 1, but the other (0.8454149542619667) is in GIPSYNode 3.

Resource Demand Information			
Demand Type	Demand Signature	Request Resource	Resource ID
Resource	129e1278-4379-411f-9346-160d985f7edc	GEER	0.8454149542619667
Resource	15c88c47-2258-4fae-95cb-494204ee1a2e	GEER	0.8474930470440531

4.4.4.1 Configuration

- On GIPSYNode 1

  1. Create two `GEER`s, based on these two `GEERSignature`, generate two `ResourceDemand` (the purpose of the resource demands are requesting the `GEER`).
  2. Put one `GEER` in the `LocalGEERPool` of DGT in GIPSYNode 1.

- On GIPSYNode 2

1. Run the DST.

- On GIPSYNode 3

1. Create one DGT in GIPSYNode 3, put the second `GEER` in its `LocalGEERPool`.

4.4.4.2 Workflow

The workflow for resource demand processing (refer to Figure 30) should be as follows:

1. The `DepositDemand` of the DGT retrieves pending demand from `LocalDemandStore`, since it is a resource demand, it checks its local GEER pool for the requested GEER. If the GEER is in the local GEER pool, get the GEER and set it as the result of the resource demand, set the demand to the computed state, and put it back to the local demand store, then send the result to the DST.

2. If the requested GEER is not archived in the local GEER pool, the pending demand is sent to the DST and then and put it back to the local demand store as an inprocess demand.

3. The `WithdrawDemand` of the other DGT gets a pending demand from the DST and checks its own local GEER pool, once it gets the GEER from it, it fills the demand's result field with the GEER, and puts the demand back to the DST.

4. The `WithdrawResult` of the first DGT gets an inprocess demand from its own local demand store and checks the DST o see if this demand has been set to the computed sate, if so, it gets the computed demand and puts it in its local demand store.

4.4.4.3 Testing Result

1. The demands in the local demand store at Point 1:

Demand Type	Demand Signature	Resource ID	Demand State
RESOURCE	129e1278-4379-411f-9346-160d985f7edc	0.8454149542619667	PENDING
RESOURCE	15c88c47-2258-4fae-95cb-494204ee1a2e	0.8474930470440531	PENDING

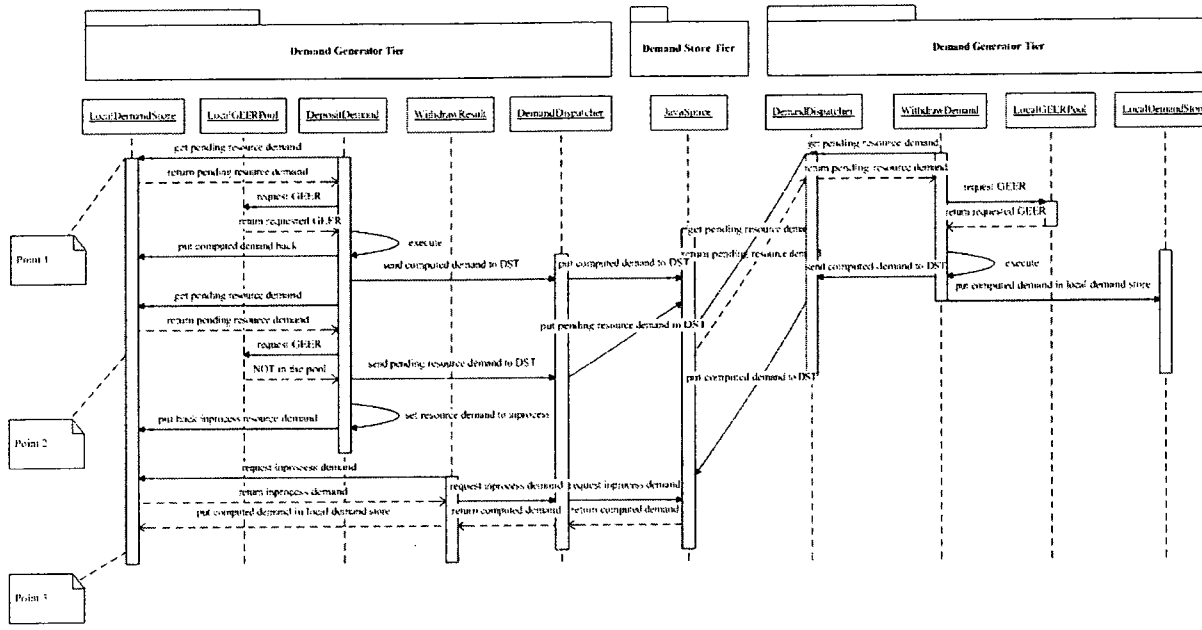

Figure 30: Resource Demand Processing

2. The demands in the local demand store at Point 2, the GEER (0.8474930470440531) is in local GEER pool:

Demand Type	Demand Signature	Resource ID	Demand State
RESOURCE	129e1278-4379-411f-9346-160d985f7edc	0.8454149542619667	PENDING
RESOURCE	15c88c47-2258-4fae-95cb-494204ee1a2e	0.8474930470440531	COMPUTED

3. The demands in the local demand store at Point 3, the resource demand for GEER (0.8454149542619667) has been computed by the other DGT:

Demand Type	Demand Signature	Resource ID	Demand State
RESOURCE	129e1278-4379-411f-9346-160d985f7edc	0.8454149542619667	COMPUTED
RESOURCE	15c88c47-2258-4fae-95cb-494204ee1a2e	0.8474930470440531	COMPUTED

4.5 Summary

Since our implementation involves both refactoring the existing code and adding the new framework in an ongoing project, we need to make sure that both the new code is working and do not introduce

any side effect when working with the previous code. In order to achieve that, we applied test-driven development. Thus, the implementation and testing are performed together and presented in one chapter and the layout of this chapter follows the integration of the actual development. In Section 4.2 we introduced the concept of multi-tier and the implementation framework. In Section 4.3 we mentioned how we worked with the previous code, and in Section 4.4 we presented the multi-tier test cases in the GIPSY environment.

In the course of the integration testing for intensional demand, procedural demand, and the resource demand, we proved that the multi-tier runtime system has been integrated into the GIPSY project and is capable of properly migrating the various existing demands in a distributed, demand-driven manner.

In the next chapter, we will present the conclusion of this research according to our original goals, as well as to present limitations of our contributions, and future work arising from our research work.

Chapter 5

Conclusions and Future Work

"The skill of writing is to create a context in which other people can think."

Edwin Schlossberg

In this chapter, we provide conclusions on our research contributions and also expose further work on the existing multi-tier architecture and farther in the GIPSY project in general.

5.1 Conclusion

5.1.1 Research Contributions

- In Chapter 3, we overviewed the Lucid programming language and the GIPSY project, in order to present the design rationale as to why a distributed and multi-tier architecture is needed. Then we analyzed the workflow of demand migration in a demand-driven distributed execution engine for GIPSY and setup the tasks for each tier. Later in this chapter, we proposed our development methodology.

- In Chapter 4, we presented the implementation that we carried out, including the refactoring of the existing demand migration system and wrapped it into our multi-tier runtime system, then we integrated all the newly developed multi-tier architecture in the GIPSY project.

- Made demand-driven execution and multi-tier architecture as the main concepts of GIPSY distributed execution environment. Provided the building blocks that allows the monitoring

of GISPY networks in the future.

- The `Configration` class was introduced, which contains serializable configuration of this GIPSY instance and components. We have shown that it can effectively be used to setup our integration testing environment. The configuration obects can be used for static and run-time configuration management. In the future, this can the base for the implementation of *System Demand* and the Management Tier.

- The `Demand` class has been developed and made the internal data structure of the runtime system. The parameter of `DemandSignature`, `DemandState`, and `DemandType` provides demands full flexibility to implement the various execution workflows that we have tested.

- From the perspective of the GIPSY project, this thesis is a step forward based on the previous research work on the demand migration framework, which now has been wrapped in the multi-tier framework and connected with the front end of GIPSY. Also, concepts such as the local demand store and the local GEER pool enhanced the rationality of the demand execution workflow.

5.2 Future Work & Limitations

We have achieved the implementation of the most part of a distributed run-time system for the GIPSY. We emphasize that the system is integrated and running according to our purposed workflow, but there are still limitations that need to be addressed in the future.

- Now the system is running with a `DemandFacotry` (the demand generator simulator) which generates demands statically. In the future we need the new eduction engine and demand generator, which is currently being re-designed. Once finished, it will be easily integrated, as our demand simulator uses the same interface as the eduction engine.

- As stated in the thesis, the runtime system of GIPSY is demand-driven, the granularity of demand can be one of the fundamental issues that decides its performance and workflow. Certain research investment is needed to address what is the most reasonable and efficient solution in order to reach for qualities such as scalability.

- Now the system is deployed manually and working strictly following pre-fixed configurations. The development of a management tier that is able to analyze the workload of each tier and automatically create or remove tiers would improve the flexibility, performance and scalability of the system. For that, the introduction of `system demand` should be necessary.

- Based on the current work, there are several similarities between DGT and DWT. They both use buffer objects and except for their purpose, their interaction workflow are much alike. So in the long run, multi-tier may become homogeneous, where each node can act as either of the tiers as descibed in the multi-tier architecture (i.e. each node can perform the task of generator, worker, store, or even management tier when necessary).

Bibliography

[AFJW95] Edward A. Ashcroft, Anthony Faustini, Raganswamy Jagannathan, and William W. Wadge. *Multidimensional, Declarative Programming.* Oxford University Press, London, 1995.

[Agi95] Iskender Agi. GLU for multidimensional signal processing. In *ISLIP'95: The 8th International Symposium on Languages for Intensional Programming*, Sydney, Australia, 1995.

[AW76] Edward A. Ashcroft and William W. Wadge. Lucid – a formal system for writing and proving programs. *SIAM J. Comput.*, 5(3), 1976.

[AW77a] Edward A. Ashcroft and William W. Wadge. Erratum: Lucid – a formal system for writing and proving programs. *SIAM J. Comput.*, 6(1):200, 1977.

[AW77b] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[AW82] Edward A. Ashcroft and William W. Wadge. *R* for semantics. *ACM Transactions on Programming Languages and Systems*, 4(2):283–294, April 1982.

[BddzzP+10] Brian Berliner, david d 'zoo' zuhn, Jeff Polk, Larry Jones, Derek Robert Price, Mark D. Baushke, et al. Concurrent Versions System (CVS). [online], 1989–2010. `http://savannah.nongnu.org/projects/cvs/`.

[Car47] Rudolf Carnap. *Meaning and Necessity: a Study in Semantics and Modal Logic.* University of Chicago Press, Chicago, USA, 1947.

[CB+00] R. G. G. Cattell, D. K. Barry, et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 2000. ISBN 0201633612.

[Day95] J. Day. The (un)revised OSI reference model. *SIGCOMM Comput. Commun. Rev.*, 25(5):39–55, 1995.

[Du91] W. Du. *Indexical Parallel Programming.* PhD thesis, Department of Computer Science, Victoria University, Canada, 1991.

[DW90a] Weichang Du and William W. Wadge. A 3D spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, June 1990.

[DW90b] Weichang Du and William W. Wadge. The eductive implementation of a three-dimensional spreadsheet. *Software Practice and Experience*, 20(11):1097–1114, November 1990.

[DWP81] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics.* D. Reidel, Dordrecht, The Netherlands, 1981.

[E+10] Eclipse contributors et al. Eclipse Platform. eclipse.org, 2000–2010. `http://www.eclipse.org`, last viewed February 2010.

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring, Improving the Design of Existing Code.* Addison-Wesley, 2000.

[FL89] Anthony A. Faustini and E. B. Lewis. *Towards a Real-Time Dataflow Language.* IEEE Computer Society, Los Alamitos, CA, USA, 1989.

[Fle01] R. Flenner. *Jini and JavaSpaces Application Development.* Sams, 2001.

[FW87] Antony A. Faustini and William W. Wadge. An eductive interpreter for the language Lucid. *SIGPLAN Not.*, 22(7):86–91, 1987.

[Gal75] D. Gallin. *Intensional and Higher-Order Modal Logic: With Applications to Montague Semantics.* North-Holland, Amsterdam, The Netherlands, 1975.

[GB10] Erich Gamma and Kent Beck. JUnit. [online], Object Mentor, Inc., 2001–2010. `http://junit.org/`.

[GMP05] Peter Grogono, Serguei Mokhov, and Joey Paquet. Towards JLucid, Lucid with embedded Java functions in the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 15–21. CSREA Press, June 2005.

[GP70] Richard Gauthier and Stephen Pont. *Designing Systems Programs.* Prentice-Hall, 1970.

[JBo07] JBoss. JBoss application server guide. [online], 2007. `http://www.jboss.org/products/jbossas`.

[JD96] Raganswamy Jagannathan and Chris Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.

[JDA97] Raganswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A high-level system for granular data-parallel programming. In *Concurrency: Practice and Experience*, volume 1, pages 63–83, 1997.

[Jin07] Jini Community. Jini network technology. [online], September 2007. `http://java.sun.com/developer/products/jini/index.jsp`.

[Kri66] Saul A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 31(2):276–277, 1966.

[Kri69] Saul A. Kripke. Semantical considerations on modal logic. *Journal of Symbolic Logic*, 34(3):501, 1969.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Lu04] Bo Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages.* PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, March 2004.

[Mam05] Qusay H. Mamoud. Getting started with JavaSpaces technology: Beyond conventional distributed programming paradigms. [online], July 2005. `http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/`.

[Mok05] Serguei A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN 0494102934; online at `http://arxiv.org/abs/0907.2640`.

[Mok10a] Serguei A. Mokhov. GIPSY naming & coding conventions, 2003–2010. `http://newton.cs.concordia.ca/~gipsy/cgi-bin/viewcvs.cgi/*checkout*/resources/doc/conventions/coding/index.html`.

[Mok10b] Serguei A. Mokhov. *Hybrid Intensional Computing in GIPSY: JLucid, Objective Lucid and GICF.* Lambert Academic Publishing, March 2010. ISBN 978-3-8383-1198-2.

[MP05] Serguei Mokhov and Joey Paquet. General imperative compiler framework within the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 36–42. CSREA Press, June 2005.

[MP10] Serguei A. Mokhov and Joey Paquet. Using the General Intensional Programming System (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions. In *Proceedings of SERA 2010*, pages 101–109. IEEE Computer Society, May 2010. Online at `http://arxiv.org/abs/0906.3911`.

[Ost81] C. B. Ostrum. *The Luthid 1.0 Manual.* Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[Paq99] Joey Paquet. *Scientific Intensional Programming.* PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[Paq09] Joey Paquet. Distributed eductive execution of hybrid intensional programs. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pages 218–224, Seattle, Washington, USA, July 2009. IEEE Computer Society.

[PGW04] Joey Paquet, Peter Grogono, and Ai Hua Wu. Towards a framework for the general intensional programming compiler in the GIPSY. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, October 2004. ACM.

[PK00] Joey Paquet and Peter Kropf. The GIPSY architecture. In *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[PK04] Nikolaos S. Papaspyrou and Ioannis T. Kassios. GLU# embedded in C++: a marriage between multidimensional and object-oriented programming. *Softw., Pract. Exper.*, 34(7):609–630, 2004.

[PKL93] John Plaice, Ridha Khedri, and Rene Lalement. From abstract time to real time. In *In Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 83–93, 1993.

[PMDW08] John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential demand-driven evaluation of eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1266–1271, Turku, Finland, July 2008. IEEE Computer Society.

[PMT08] Joey Paquet, Serguei A. Mokhov, and Xin Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[Pou08] Amir Hossein Pouteymour. Comparative study of Demand Migration Framework implementation using JMS and Jini. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, September 2008.

[PP94] Joey Paquet and John Plaice. On the design of an indexical query language. In *Proceedings of the Seventh International Symposium on Lucid and Intensional Programming*, pages 28–36, 1994.

[PRP06] Arno Puder, Kay Romer, and Frank Pilhofer. *Distributed systems architecture: a middleware approach.* Elsevier, 2006. ISBN: 9781558606487.

[PVP07] Amir Hossein Pourteymour, Emil Vassev, and Joey Paquet. Towards a new demand-driven message-oriented middleware in GIPSY. In *Proceedings of PDPTA 2007*, pages 91–97, Las Vegas, USA, June 2007. PDPTA, CSREA Press.

[PW93] John Plaice and William W. Wadge. A new approach to version control. *IEEE Transactions on Software*, 19(3):268–276, March 1993.

[PW05] Joey Paquet and Ai Hua Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14. CSREA Press, June 2005.

[Ren02] Chun Lei Ren. General intensional programming compiler (GIPC) in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.

[RP08] Toby Rahilly and John Plaice. A multithreaded implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1272–1277, Turku, Finland, July 2008. IEEE Computer Society.

[Sim99] Charles Simonyi. Hungarian notation. [online], November 1999. `http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx`.

[Sun06] Sun Microsystems, Inc. The J2EE 1.4 tutorial: Types supported by JAX-RPC. Sun Microsystems, Inc., 2006. `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXRPC4.html#wp130550`.

[Sun07] Sun Microsystems, Inc. Java Message Service (JMS). [online], September 2007. `http://java.sun.com/products/jms/`.

[SW08] James Shore and Shane Warden. *The Art of Agile Development.* O'Reilly, 2008. ISBN: 0-596-52767-5.

[Tao94] S. Tao. *TLucid and Intensional Attribute Grammars.* PhD thesis, Department of Computer Science, Victoria University, Canada, 1994.

[Tao04] Lei Tao. Warehouse and garbage collection in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[The10] The hsqldb Development Group. HSQLDB – lightweight 100% Java SQL database engine v.1.8.1.2. [digital], 2001–2010. `http://hsqldb.org/`.

[Ton08] Xin Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, April 2008.

[Vas05] Emil Iordanov Vassev. General architecture for demand migration in the GIPSY demand-driven execution engine. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, June 2005. ISBN 0494102969.

[vB88] J. van Benthem. *A Manual of Intensional Logic.* CSLI Publications, Stanford and The University of Chicago Press, 1988.

[VP05a] Emil Vassev and Joey Paquet. A general architecture for demand migration in a demand-driven execution engine in a heterogeneous and distributed environment. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference (CNSR 2005)*, pages 176–182. IEEE Computer Society, May 2005.

[VP05b] Emil Vassev and Joey Paquet. A generic framework for migrating demands in the GIPSY's demand-driven execution engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 29–35. CSREA Press, June 2005.

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.

[Wad03] William W. Wadge. Hammings problem example. [online], December 2003. `http://i.csc.uvic.ca/home/hei/lup/contents.html`.

[Wan06] Kaiyu Wan. *Lucx: Lucid Enriched with Context.* PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[WAP05] Kaiyu Wan, Vasu Alagar, and Joey Paquet. Lucx: Lucid enriched with context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 48–14. CSREA Press, June 2005.

[WBSY98] William W. Wadge, G. Brown, M. C. Schraefel, and T. Yildirim. Intensional HTML. In *4th International Workshop PODDP'98*, March 1998.

[WPM07] Aihua Wu, Joey Paquet, and Serguei A. Mokhov. Object-Oriented Intensional Programming: A New Concept in Object-Oriented and Intensional Programming Domains. Unpublished, 2007.

[WPM08] Aihua Wu, Joey Paquet, and Serguei A. Mokhov. Object-Oriented Intensional Programming: Intensional Classes Using Java and Lucid. Unpublished, 2008.

[WPM10] Aihua Wu, Joey Paquet, and Serguei A. Mokhov. Object-oriented intensional programming: Intensional Java/Lucid classes. In *Proceedings of SERA 2010*, pages 158–167. IEEE Computer Society, 2010. Online at: `http://arxiv.org/abs/0909.0764`.

[Wu02] Ai Hua Wu. Semantic checking and translation in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. *SML Technical Report Series*, 1994.

Appendix A

Agile Software Development

A.1 Agile Development

Agile development is a way of thinking about software development. The canonical description of this way of thinking is the Agile Manifesto, a collection of 4 values in Appendix A.4 and 12 principles in Appendix A.5. To "be agile", you need to put the agile values and principles into practice.

Agile methods are process that support the agile development. Examples include *Extreme Programming* and *Scrum* [SW08].

Agile methods consist of individual elements called *practices.* Practices include using version control, setting coding standards, and giving weekly demos to your stakeholders. Most of these practices have been around for years. Agile methods combine them in unique ways and mixing in a few new ideas. The result is a lean, powerful, self-reinforcing package.

A.2 Extreme Programming

One of the most astonishing premises of XP is that you can eliminate requirements, design and testing phases as well as the formal documents that go with them (XP lifecycle is shown in Figure 31 [SW08]).

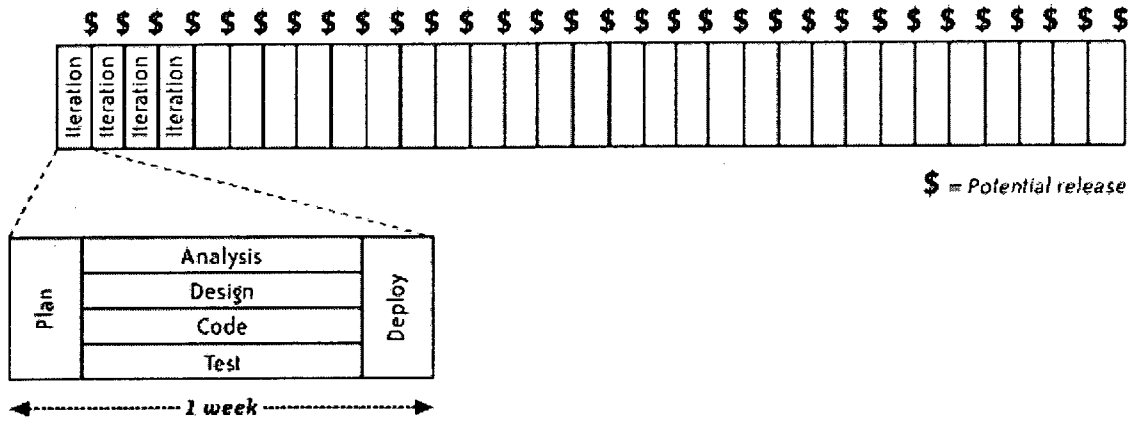

Figure 31: Extreme Programming Lifecycle

A.3 Adopting XP Features

As this thesis involves both implementation and refactoring to integrate previous code into the new design. We would like to adopt a subset of XP practices.

A.3.1 Coding Convention

Coding Convention are the guidelines to which all developers agree to adhere when programming includes: *Programming Language*, *Integrated Development Environment*, *Naming Convention*, *File and directory layout*, and *Coding Standards*.

A.3.2 Version Control

A *version control system* provides a central repository that helps coordinate changes to files and also provides a history of changes. Our GIPSY team use *Concurrent Version System* (CVS) [BddzzP+10] to manage the source code. CVS allows multiple developers work on the up-to-date source tree in parallel that keeps tracks of the revision history and works in an transactional manner. It is an orderly process in which our team members get the latest code from the server, do the work, run all the tests to confirm the code works, then check in the changes. This continuous integration process occurs several times every iteration.

A.3.3 Iteration Planning

At the beginning of an iteration, we hold a team meeting, normally with brainstorm to identify the most valuable tasks for this iteration. By the end of the iteration, members carried out working, tested software for each task and are ready to begin the cycle again.

A.3.4 Test-Driven Development

Test-Driven Development (TDD), is a rapid cycle of testing, coding and refactoring. TDD uses an approach similar to double-entry bookkeeping. The programmer communicates his or her attention twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere.

In GIPSY, we use *JUnit* [GB10] to write and run automated repeatable unit tests. With the test cases, we are able to spend little time debugging and find mistakes very quickly and have little difficulty fixing them.

A.3.5 Refactoring

Refactoring is the process of changing the design of code without changing its behavior. In this thesis we integrated the existing code into a new architecture, so not only analyzing the existing code, separating its behavior is also needed. We will introduce more about the delivered refactoring in Chapter 4.

A.4 Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value [SW08]:

- ***Individuals and interactions*** over processes and tools
- ***Working software*** over comprehensive documentation
- ***Customer collaboration*** over contract negotiation
- ***Responding to change*** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Bech	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brain Marich	

A.5 Principles behind the Agile Manifesto

We follow the these principles: [SW08]

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile process harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motived individuals. Given them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile process promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity, the art of maximizing the amount of work not done, is essential.
11. The best architecture, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Appendix B

Coding Conventions Applied in GIPSY

Programming Language The primary implementation language of GIPSY is Java. We have chosen Java to implement our project using the Java programming language mainly because of its portability as well as facilities, for example, memory management and multi-thread programming, so we can concentrate more on the algorithms instead. Java also provides built-in types and data-structures to management collections efficiently.

Integrated Development Environment (IDE) Eclipse [E+10] is an open source IDE for Java project development, its community provides many extended tools for development, refactoring and deployment. GIPSY use Eclipses as the implementation platform.

Naming Convention Hungarian notation [Sim99] is a naming convention in computer programming invented by Charles Simonyi, in which the name of a variable indicates its type or intended use. In GIPSY we apply simplified hungarian notation [Mok10a] in order to achieve consistency and uniform style in our project. For detailed coding convention for GIPSY please refer to Appendix C.

File and directory layout As in Java, a fully qualified class name includes all the packages starting from the top level directory of the hierarchy all the way down to the class itself, separated

by a dot. The structure was performed in correspondence with the original conceptual design primarily produced by Dr. Joey Paquet, Dr. Bo Lu, and Dr. Aihua Wu. The basic structure is as follows. The top root hierarchy is logically the `gipsy` package. The major non-utility packages under it, which comes from the conceptal design, are `GIPC`, `GEE`, `RIPE`, and `tests`. Under the `GIPC` package the major modules include `Preprocessing` for general GIPSY program preprocessing, `intensional` and `imperative` language compilers and their necessary followers. The `GEE` package includes `IDP` for demand propagation and `IVM` for caching and garbage collection. Our multi-tier design will be located in `GEE` package as well. Under `RIPE` we have interactive runtime editing and monitoring modules that include textual editor, DFG editor, and the web-based editor.

Coding Standards When our GIPSY team agree on coding standards, we improve the maintainability and readability of the code, but for the legacy code that does not fit the standard, we are not going to fix it until we have to touch it.

Appendix C

Naming & Coding Conventions

C.1 Hungarian Notation

Microsoft's Chief Architect Dr. Charles Simonyi introduced an identifier naming convention that adds a prefix to the identifier name to indicate the functional type of the identifier. This system became widely used inside Microsoft. It came to be known as "Hungarian notation" [Sim99] because the prefixes make the variable names look a bit as though they're written in some non-English language and because Simonyi is originally from Hungary.

Hungarian names have two parts. The prefix contains type information specifying what this thing is, while the suffix contains descriptive information specifying what it means. The prefix is composed of standard elements that are the same for any program you write in a given language. The suffix is composed of English words that tend to be application-specific.

C.2 Simplified Hungarian Notation for the Team

This proposed naming scheme is supposed to be used by everyone programming in Java to maintain consistency among the code base for the projects where the Java programming language is used.

C.2.1 Class Names

1. All names of all classes start with the capital letter;

2. If the class name is a multiword name, every single word must be capitalized;

C.2.2 Method Names

1. First letter of every user-defined function is in lower case;
2. If the function's name has multiple words every other word should be capitalized;
3. For getters and setters (class methods that set or return data members of the class) should have get and set prefixes, and after property name starting with capital letter;
4. Name of the function has to have a meaning, obviously.

C.2.3 Variable Names

1. Each variable starts with a special prefix indicating variable's scope.
2. The next part of the variable name is its type. This is especially useful for the languages where the data type does not really exists like PHP, PERL, and JavaScript, and it might not be clear from the variable's declaration what type of value it has. In addition, specifying the type improves readability of the code and reduces number of bugs related to misinterpretation of the variable's value type.

C.3 Other Coding Conventions

Other coding conventions including Class Structure, Bracketing, Indentation & Spacing, Comments and etc. All the details please refer to GIPSY coding convention page [Mok10a].

Appendix D

Procedure for Setup Jini & JMS Running Environment

This document is summarized by Yi Ji, covers how to setup and run Jini & JMS running environment.

D.1 Setup of Jini Environment

This section introduces the steps to compile and run the Jini code in the GIPSY project in Windows XP. To use this guide, readers are required to have basic Java and Eclipse experience, basic Jini knowledge (for example, service, lookup, JavaSpace, etc), and the basic understanding of the GIPSY project structure.

D.1.1 Get and install the appropriate software, and import the GIPSY project

1. JDK 6 update 14 or later `http://java.sun.com/javase/downloads/index.jsp`. It is better to set the JAVA_HOME environment variable.

2. Eclipse IDE for Java Developers `http://www.eclipse.org/downloads/`. Unpack the IDE, open it and import the GIPSY project from the GIPSY CVS into the eclipse.

3. Jini Technology Starter Kit v2.1 `http://java.sun.com/developer/products/jini/index.jsp`. The installation should require no administrator accounts. The term JINI_HOME would be used in this manual to refer the installation directory.

D.1.2 Compile the Jini code of the GIPSY project

1. Copy the jini-core.jar and jini-ext.jar from JINI_HOME /lib into the gipsy/lib.
2. Configure the project Build Path by adding the above jars inside lib through "Add JARs".
3. Build the project automatically or manually.

D.1.3 Start the Jini service

1. Go to JINI_HOME/installverify, and double-click the Launch-All shortcut. The shortcut should launch a service window and a Service Browser window.
2. In the Service Browser window, click the "Registrar", and select the register representing your computer. Then there would be 6 services appear in the "Matching Service" area, including JavaSpace05, LookupDiscoveryService, ServiceRegistrar and TransactionManager.
3. Leave the two windows open and to not touch them unless you want to shut down all the services.

D.2 Setup of JMS Environment

This section introduces the steps to compile and run JMS code in the GIPSY project in Windows XP. To use this guide, readers are required to have basic Java and Eclipse experience, basic JMS knowledge (for example, to know what a queue or a broker is, etc), and the basic understanding of the GIPSY project structure.

D.2.1 Get and install the appropriate software, and import the GIPSY project

1. JDK 6 update 14 or later `http://java.sun.com/javase/downloads/index.jsp`. It is better to set the JAVA_HOME environment variable.

2. Eclipse IDE for Java Developers `http://www.eclipse.org/downloads/`. Unpack the IDE, open it and import the GIPSY project from the GIPSY CVS into the eclipse.

3. Open Message Queue 4.2 Binary downloads `https://mq.dev.java.net/downloads.html`. This version does not require an installer; you could simply un-zip the file. But a system or user environment variable called IMQ_JAVAHOME needs to be set to the current JRE directory before you run it. The Open Message Queue installation directory will be called MQ_HOME in this manual.

D.2.2 Compile the JMS code of the GIPSY project

1. Copy the fscontext.jar, jms.jar and imq.jar from MQ_HOME/lib into the gipsy/lib

2. Configure the project Build Path by adding the above jars inside lib through "Add JARs"

3. Build the project automatically or manually.

D.2.3 Set up the message queue service

1. Go to MQ_HOME/bin, and double-click the imqbrokerd.exe. The imqbrokerd.exe file would then open a command window, and launch a default message queue broker instance called "imqbroker", providing message queue services. To be more specific, the "imqbroker" instance would reside in the MQ_HOME/var/instances/ directory.

2. Still in the MQ_HOME/bin, double-click the imqadmin.exe. The imqadmin.exe would then launch a Sun Java. System Message Queue Administration Console. Unless otherwise specified, the following points are performed in the Administration Console.

3. In the Administration Console, if there is no "MyBroker" under the "Brokers", then right-click "Brokers" and choose "Add Broker", and enter the Broker Label as "MyBroker", Host as "localhost", Primary Port "7676", and Username "admin". Then click "OK".

4. Right-click the "MyBroker" and choose "Connect to Broker", then a login window should appear with the Username "admin". Enter the Password as "admin", which is the default password, and click "OK".

5. Go to the "Destinations" under "MyBroker" to see if there are queues called "demands" and "results". If there are not, right-click the "Destinations", and choose "Add Broker Destination". In the pop-up window, enter the Destination Name as "demands", choose the Type "Queue" and then do not touch anything except setting the Max Number of Active Consumers to "100". Click "OK" then a queue called "demands" should appear in the "Destinations". Similarly add another queue called "results".

6. Now we set up message queue administration objects. Go to the "Object Stores" which is in the same level as "Brokers". Check if there is a "MyObjectStore". If none, right-click "Object Store" and choose "Add Object Store". In the pop-up window enter the Object Store Label as "MyObjectStore", set the Naming Service Properties:
   *"java.naming.factory.initial" value "com.sun.jndi.fscontext.RefFSContextFactory"
   *"java.naming.provider.url" value "file:///g:/mqcf" This value is the folder directory where the administration objects will be stored. The client program would use the directory to fetch these objects and use them to connect to the queue service.
   Click "OK".

7. Go to the "Destinations" under the "MyObjectStore", and right-click the "Destinations" and choose "Add Destination Object". In the pop-up window enter the Lookup Name "MyQueue", choose the Type "Queue", enter the Destination Name "demands" and click "OK". Now an entry with Lookup Name "MyQueue" pointing to the real "demands" queue should appear. Similarly add another lookup entry with LookupName "result" pointing to the queue "results".

8. Go to the "Connection Factories" under the "MyObjectStore", right-click it and choose "Add

Connection Factory Object". In the pop-up window enter the Lookup Name "MyQueue-ConnectionFactory", choose the Type "QueueConnectionFactory", and in the "Connection Handling" tab enter the Message Server Address List "your IP address".

9. Now close the Administration Console and open the file gipsy/jms.config to check if the 5 entry values are exactly the same as you set. You could either leave the imqbrokerd.exe window or close it.

D.2.4 Start the message queue service and run the code

1. If the imqbrokerd.exe is not running, go to the MQ_HOME/bin folder and double-click it.
2. Open the GIPSY project from eclipse. Make sure that the jms.config file is in the root folder.
3. Run the classes within the same groups mentioned above with the "main()" method.

Appendix E

Result for Procedural Demand

E.1 Result for IP Request

132.205.99.86

E.2 Result for PI Calculation

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065643086021394946395224737190702179860943702770539217176293176752384674818467669405132000568127145263560827785771342757789609173637178721468440901224953430146549585371050792279689258923542019956112129021960864034418159813629774771309960518707211349999998372978049951059731732816096318595024459455346908302642522308253344685035261931188171010003137838752886587533208381420617177669147303598253490428755468731159562863882353787593751957781857780532

17122680661300192787661119590921642019893809525720106548586327886593615338

18279682303019520353018529689957736225994138912497217752834791315155748572

42454150695950829533116861727855889075098381754637464939319255060400927701

67113900984882401285836160356370766010471018194295559619894676783744944825

53797747268471040475346462080466842590694912933136770289891521047521620569

66024058038150193511253382430035587640247496473263914199272604269922796782

35478163600934172164121992458631503028618297455570674983850549458858692699

56909272107975093029553211653449872027559602364806654991198818347977535663

69807426542527862551818417574672890977772793800081647060016145249192173217

21477235014144197356854816136115735255213347574184946843852332390739414333

45477624168625189835694855620992192221842725502542568876717904946016534668

04988627232791786085784383827967976681454100953883786360950680064225125205

11739298489608412848862694560424196528502221066118630674427862203919494504

71237137869609563643719172874677646575739624138908658326459958133904780275

90099465764078951269468398352595709825822620522489407726719478268482601476

99090264013639443745530506820349625245174939965143142980919065925093722169

64615157098583874105978859597729754989301617539284681382686838689427741559

91855925245953959431049972524680845987273644695848653836736222626099124608

05124388439045124413654976278079771569143599770012961608944169486855584840

63534220722258284886481584560285060168427394522674676788952521385225499546

66727823986456596116354886230577456498035593634568174324112515076069479451

09659609402522887971089314566913686722874894056010150330861792868092087476

09178249385890097149096759852613655497818931297848216829989487226588048575

64014270477555132379641451523746234364542858444795265867821051141354735739

52311342716610213596953623144295248493718711014576540359027993440374200731

05785390621983874478084784896833214457138687519435064302184531910484810053

70614680674919278191197939952061419663428754440643745123718192179998391015

91956181467514269123974894090718649423196156794520809514655022523160388193

01420937621378559566389377870830390697920773467221825625996615014215030680

38447734549202605414665925201497442850732518666002132434088190710486331734

649651453905796268561005508106658796998163574736384052571459102897064140110971206280439039759515677157700420337869936007230558763176359421873125147120532928191826186125867321579198414848829164470609575270695722091756711672291098169091528017350671274858322287183520935396572512108357915136988209144210067510334671103141267111369908658516398315019701651511685171437657618351556508849099898599823873455283316355076479185358932261854896321329330898570642046752590709154814165498594616371802709819943099244889575712828905923233260972997120844335732654893823911932597463667305836041428138830320382490375898524374417029132765618093773444030707469211201913020330380197621101100449293215160842444859637669838952286847831235526582131449576857262433441893039686426243410773226978028073189154411010446823252716201052652272111660396665573092547110557853763466820653109896526918620564769312570586356620185581007293606598764861179104533488503461136576867532494416680396265797877185560845529654126654085306143444318586769751456614068007002378776591344017127494704205622305389945613140711270004078547332699390814546646458807972708266830634328587856983052358089330657574067954571637752542021149557615814002501262285941302164715509792592309907965473761255176567513575178296664547791745011299614890304639947132962107340437518957359614589019389713111790429782856475032031986915140287080859904801094121472213179476477726224142548545403321571853061422881375850430633217518297986622371721591607716692547487389866549494501146540628433663937900397692656721463853067360965712091807638327166416274888800786925602902284721040317211860820419000422966171196377921337575114959501566049631862947265473642523081770367515906735023507283540567040386743513622224771589150495309844489333096340878076932599397805419341447377441842631298608099888687413260444493414473774418426312986080998886874132604444

Index